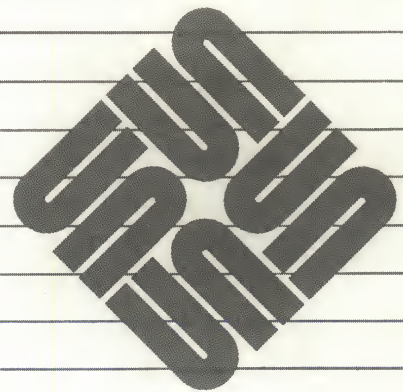




SunView™ 1 System Programmer's Guide



Sun Workstation and the Sun logo
are trademarks of Sun Microsystems, Incorporated.

SunView is a trademark of Sun Microsystems, Incorporated.

UNIX[®] is a registered trademark of AT&T.

All other products or services mentioned in this document are
identified by the trademarks or service marks of their
respective companies or organizations.

Copyright © 1982, 1983, 1984, 1985, 1986, 1987, 1988 by Sun Microsystems, Inc.

This publication is protected by Federal Copyright Law, with all rights reserved. No part of this publication may be reproduced, stored in a retrieval system, translated, transcribed, or transmitted, in any form, or by any means manual, electric, electronic, electro-magnetic, mechanical, chemical, optical, or otherwise, without prior explicit written permission from Sun Microsystems.

Contents

Chapter 1 Introduction	3
What is SunView?	3
Changes From Release 2.0	3
Organization of Documentation	3
Compatibility	3
Changes in Release 4.0	4
 Chapter 2 Overview	 7
2.1. SunView Architecture	7
2.2. Document Outline	7
 Chapter 3 SunView System Model	 11
3.1. A Hierarchy of Abstractions	11
Data Managers	13
Data Representations	13
3.2. Model Dynamics	13
Tiles and the Agent	14
Windows	14
Desktop	15
Locking	16
Colormap Sharing	16
Workstations	17
 Chapter 4 The Agent & Tiles	 21

4.1. Registering a Tile With the Agent	21
Laying Out Tiles	22
Dynamically Changing Tile Flags	23
Extracting Tile Data	23
4.2. Notifications From the Agent	23
4.3. Posting Notifications Through the Agent	24
4.4. Removing a Tile From the Agent	26
Chapter 5 Windows	29
5.1. Window Creation, Destruction, and Reference	29
A New Window	29
An Existing Window	30
References to Windows	30
5.2. Window Geometry	31
Querying Dimensions	31
The Saved Rect	32
5.3. The Window Hierarchy	32
Setting Window Links	32
Activating the Window	33
Defaults	33
Modifying Window Relationships	34
Window Enumeration	35
Enumerating Window Offspring	35
Fast Enumeration of the Window Tree	36
5.4. Pixwin Creation and Destruction	36
Creation	36
Region	37
Retained Image	37
Bell	37
Destruction	37
5.5. Choosing Input	37
Input Mask	37
Manipulating the Mask Contents	38

Setting a Mask	38
Querying a Mask	39
The Designee	39
5.6. Reading Input	39
Non-blocking Input	39
Asynchronous Input	39
Events Pending	40
5.7. User Data	40
5.8. Mouse Position	40
5.9. Providing for Naive Programs	41
Which Window to Use	41
The Blanket Window	41
5.10. Window Ownership	42
5.11. Environment Parameters	42
5.12. Error Handling	43
Chapter 6 Desktops	47
Look at sunview	47
6.1. Multiple Screens	47
The singlecolor Structure	47
The screen Structure	48
Screen Creation	48
Initializing the screen Structure	49
Screen Query	49
Screen Destruction	49
Screen Position	49
Accessing the Root FD	50
Chapter 7 Workstations	53
7.1. Virtual User Input Device	53
What Kind of Devices?	53
Void Features	54
Void Station Codes	54

Address Space Layout	54
Adding a New Segment	55
Input State Access	55
Unencoded Input	55
7.2. User Input Device Control	56
Distinguished Devices	56
Arbitrary Devices	57
Non-Vuid Devices	57
Device Removal	57
Device Query	57
Device Enumeration	58
7.3. Focus Control	58
Keyboard Focus Control	58
Event Specification	59
Setting the Caret Event	59
Getting the Caret Event	59
Restoring the Caret	59
7.4. Synchronization Control	60
Releasing the Current Event Lock	60
Current Event Lock Breaking	60
Getting/Setting the Event Lock Timeout	61
7.5. Kernel Tuning Options	61
Changing the User Actions that Affect Input	63
Chapter 8 Advanced Notifier Usage	67
8.1. Overview	67
Contents	67
Viewpoint	67
Further Information	67
8.2. Notification	68
Client Events	68
Delivery Times	68
Handler Registration	68

The Event Handler	69
SunView Usage	69
Output Completed Events	69
Exception Occurred Events	70
Getting an Event Handler	70
8.3. Interposition	72
Registering an Interposer	72
Invoking the Next Function	73
Removing an Interposed Function	74
8.4. Posting	77
Client Events	77
Delivery Time Hint	77
Actual Delivery Time	77
Posting with an Argument	78
Storage Management	78
SunView Usage	79
Posting Destroy Events	80
Delivery Time	80
Immediate Delivery	80
Safe Delivery	80
8.5. Prioritization	81
The Default Prioritizer	81
Providing a Prioritizer	81
Dispatching Events	82
Getting the Prioritizer	83
8.6. Notifier Control	85
Starting	85
Stopping	85
Mass Destruction	85
Scheduling	86
Dispatching Clients	86
Getting the Scheduler	87
Client Removal	87

8.7. Error Codes	88
8.8. Restrictions on Asynchronous Calls into the Notifier	90
8.9. Issues	91
Chapter 9 The Selection Service and Library	95
9.1. Introduction	95
9.2. Basic Concepts	96
9.3. Fast Overview	96
The Selection Service vs. the Selection Library	97
9.4. Selection Service Protocol: an Example	97
Secondary Selection Between Two Windows	97
Notes on the Above Example	101
9.5. Topics in Selection Processing	101
Reporting Function-Key Transitions	101
Sending Requests to Selection Holders	102
Long Request Replies	104
Acquiring and Releasing Selections	104
Callback Procedures: Function-Key Notifications	104
Responding to Selection Requests	105
Callback Procedures: Replying to Requests	106
9.6. Debugging and Administrative Facilities	109
9.7. Other Suggestions	109
If You Die	109
9.8. Reference Section	110
Required Header Files	110
Enumerated Types	110
Other Data Definitions	110
Procedure Declarations	112
9.9. Common Request Attributes	121
9.10. Two Program Examples	125
<i>get_selection</i> Code	125
<i>selsn_demo</i>	128
Large Selections	128

Chapter 10 The User Defaults Database	145
Why a Centralized Database?	145
10.1. Overview	146
Master Database Files	146
Private Database Files	146
10.2. File Format	147
Option Names	147
Option Values	148
Distinguished Names	148
\$Help	148
\$Enumeration	148
\$Message	149
10.3. Creating a .d File: Example	149
10.4. Retrieving Option Values	150
Retrieving String Values	150
Retrieving Integer Values	150
Retrieving Character Values	151
Retrieving Boolean Values	151
Retrieving Enumerated Values	152
Searching for Specific Symbols	152
Searching for Specific Values	153
Retrieving all Values in the Database	153
10.5. Conversion Programs	154
10.6. Property Sheets	155
10.7. Error Handling	156
Error_Action	156
Maximum_Errors	157
Test_Mode	157
10.8. Interface Summary	157
10.9. Example Program: <i>filer</i> Defaults Version	160
Chapter 11 Advanced Imaging	173
11.1. Handling Fixup	173

11.2. Icons	174
Loading Icons Dynamically	174
Icon File Format	174
11.3. Damage	176
Handling a SIGWINCH Signal	176
Image Fixup	176
11.4. Pixwin Offset Control	178
Chapter 12 Menus & Prompts	181
12.1. Full Screen Access	181
Initializing Fullscreen Mode	182
Releasing Fullscreen Mode	182
Seizing All Inputs	182
Grabbing I/O	182
Releasing I/O	182
12.2. Surface Preparation	182
Multiple Plane Groups	183
Pixel Caching	183
Saving Screen Pixels	183
Restoring Screen Pixels	184
Fullscreen Drawing Operations	184
Chapter 13 Window Management	189
Tool Invocation	190
Utilities	190
13.1. Minimal Repaint Support	192
Chapter 14 Rects and Rectlists	197
14.1. Rects	197
Macros on Rects	197
Procedures and External Data for Rects	198
14.2. Rectlists	199
Macros and Constants Defined on Rectlists	200

Procedures and External Data for Rectlists	200
Chapter 15 Scrollbars	205
15.1. Basic Scrollbar Management	205
Registering as a Scrollbar Client	205
Keeping the Scrollbar Informed	206
Handling the SCROLL_REQUEST Event	207
Performing the Scroll	208
Normalizing the Scroll	209
Painting Scrollbars	209
15.2. Advanced Use of Scrollbars	209
Types of Scrolling Motion in Simple Mode	210
Types of Scrolling Motion in Advanced Mode	211
Appendix A Writing a Virtual User Input Device Driver	215
A.1. Firm Events	215
The Firm_event Structure	215
Pairs	216
Choosing VUID Events	217
A.2. Device Controls	217
Output Mode	217
Device Instancing	217
Input Controls	218
A.3. Example	218
Appendix B Programming Notes	229
B.1. What Is Supported?	229
B.2. Library Loading Order	229
B.3. Shared Libraries vs. Shared Text	229
Shared Libraries	229
B.4. Error Message Decoding	230
B.5. Debugging Hints	230
Disabling Locking	230

B.6. Sufficient User Memory	231
B.7. Coexisting with UNIX	232
Tool Initialization and Process Groups	232
Signals from the Control Terminal	232
Job Control and the C Shell	232
Index	235

Tables

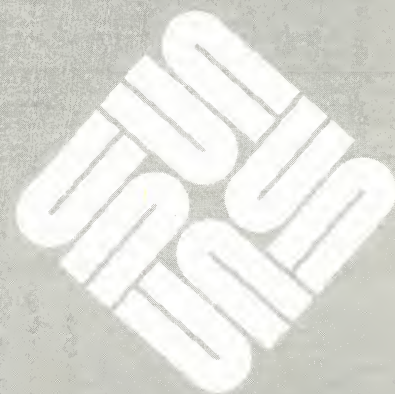
Table 9-1 Selection Service Scenario	98
Table 10-1 Defaults Metacharacters	148
Table 10-2 Default Procedures	157
Table 14-1 Rectlist Predicates	201
Table 14-2 Rectlist Procedures	202
Table 15-1 Scroll-Related Scrollbar Attributes	210
Table 15-2 Scrollbar Motions	212
Table B-1 Variables for Disabling Locking	231

Figures

Figure 3-1 SunView System Hierarchy	12
---	----

Introduction

Introduction	3
What is SunView?	3
Changes From Release 2.0	3
Organization of Documentation	3
Compatibility	3
Changes in Release 4.0	4



Introduction

What is SunView?

SunView is a system to support interactive, graphics-based applications running within windows. It consists of two major levels of functionality: the application level and the system level. The system level is described in this document and covers two major areas: the building blocks on which the application level is built and advanced application-related features.

Changes From Release 2.0

SunView is an extension and refinement of SunWindows 2.0, containing many enhancements, bug fixes and new facilities not present in SunWindows. However, the changes preserve source level compatibility between SunWindows 2.0 and SunView.

Organization of Documentation

The 2.0 *SunWindows Reference Manual* has not been reprinted for SunView.

These changes are reflected in the current organization of the SunView documentation. The material on Pixrects from the old *SunWindows Reference Manual* is in a new document titled *Pixrect Reference Manual*. Much of the functionality of the SunWindows window and tool layers has been incorporated into the new SunView interface. The basic SunView interface, intended to meet the needs of simple and moderately complex applications, is documented in the application-level manual, the *SunView 1 Programmer's Guide*.

This document is the *SunView 1 System Programmer's Guide*. It contains a combination of new and old material. Several of its chapters document new facilities such as the Notifier, the Agent, the Selection Service and the defaults package. Also included is low-level material from the old *SunWindows Reference Manual* — e.g. the window manager routines — of interest to implementors of window managers and other advanced applications.

This document is an extension of the application-level manual. You should only delve into this manual if the information in the *SunView 1 Programmer's Guide* manual doesn't answer your needs. Thus, you should read the application-level manual first.

Compatibility

Another consideration is compatibility with future releases. Most of the objects in the *SunView 1 Programmer's Guide* are manipulated through an opaque attribute value interface. Code that uses them will be more portable to future versions of SunView than if it uses the routines documented in this manual which assume particular data structures and explicit parameters. If you do use these routines then the code should be encapsulated so that low-level details are

isolated from the rest of your application.

Keep your old documentation

On the way to SunView, we have discarded documentation about the internals of some data structures that were discussed in SunWindows 2.0. In addition, we have discarded documentation about routines whose functionality is now provided by the interface discussed in the *SunView 1 Programmer's Guide*. Thus, if your application is based on the SunWindows programming interface, you should keep your 2.0 documentation. In particular, the following structures are no longer documented (there may be others): `tool`, `pixwin`, `toolsw`, and `toolio`.

Changes in Release 4.0

SunOS Release 4.0 includes further enhancements to the higher-level SunView programmatic interface documented in the *SunView 1 Programmer's Guide*, such as alerts, shadowed subframes, 'Props' actions, and so on. Sun encourages programmers to use these higher-level interfaces in preference to low-level routines documented in this manual whenever possible; this will help to insulate your applications from changes in the low-level window-system interfaces underneath SunView.

Overview

Overview	7
2.1. SunView Architecture	7
2.2. Document Outline	7



Overview

2.1. SunView Architecture

From a system point of view, SunView is a two-tiered system, consisting of the *application* and *system* layers:

- The application layer provides a set of high-level objects, including windows of different types, menus, scrollbars, buttons, sliders, etc., which the client can assemble into an application, or *tool*. This layer is sometimes referred to as the *tool layer*. The functionality provided at this level should suffice for most applications. This layer is discussed in the *SunView 1 Programmer's Guide*.
- At the *system* layer a window is presented not as an opaque object but in terms which are familiar to UNIX programmers — as a *device* which the client manipulates through a *file descriptor* returned by an *open* (2) call. This layer is sometimes referred to as the *window device layer*. The manipulation and multiplexing of multiple window devices is the subject of much of this document. The term “window device” is often shortened to just *window* in this document.

2.2. Document Outline

This document covers the follow system level topics:

- A system model which presents the levels, components and inter-relationships of the window system.
- A SunView mechanism, called the *Agent*, which includes:
 - notification of window damage and size changes.
 - reading and distribution of input events among windows within a process.
 - posting events with the Agent for delivery to other clients.
- Windows as devices, which includes:
 - reading control options such as asynchronous input and non-blocking input.
- The screen abstraction, called a *desktop*, which includes:
 - Routines to initialize new screens so that SunView may be run on them.
 - Multiple screens accessible by a single user.

- The global input abstraction, called a *workstation*, which includes:
 - environment wide input device instantiation.
 - controlling a variety of system performance and user interface options.
 - extending the *Virtual User Input Device* interface with events of your own design.
- Advanced use of the general notification-based flow of control management mechanism called the *Notifier*, which includes:
 - detection of input pending, output completed and exception occurred on a file descriptor.
 - maintenance of interval timers.
 - dispatching of signal notifications.
 - child process status and control facilities.
 - a client event notification mechanism, which can be thought of as a client-defined signal mechanism.
- The *Selection Service*, for exchanging objects and information between cooperative client, both within and between processes.
- The *defaults* mechanism, for maintaining and querying a database of user-settable options.
- Advanced imaging topics, which include:
 - the repair of damaged portions of your window, when not retained.
 - receiving window damage and size change notifications via SIGWINCH.
- The mechanisms used to violate window boundaries. You would use them if you created a menu or prompt package.
- Routines to perform *window management* activities such as open, close, move, stretch, top, bottom, refresh. In addition, there are facilities for invoking new tools and positioning them on the screen.
- Routines to manipulate individual rectangles and lists of rectangular areas. They forms what is essentially an algebra of rectangles, useful in computing window overlap, points in windows, etc.
- Advanced *icon* topics, including displaying them, accessing them from a file, their internal structure, etc..
- Advanced *scrollbar* topics, including calculating and performing your own scroll motions (in a canvas, for example).

Finally, there is an appendix on how to write a line discipline for a new input device that you want to access through SunView. Another appendix covers some programming notes.

SunView System Model

SunView System Model	11
3.1. A Hierarchy of Abstractions	11
Data Managers	13
Data Representations	13
3.2. Model Dynamics	13
Tiles and the Agent	14
Windows	14
Desktop	15
Locking	16
Colormap Sharing	16
Workstations	17



SunView System Model

3.1. A Hierarchy of Abstractions

This chapter presents the system model of SunView. It discusses the hierarchy of abstractions that make up the window system, the data representations of those abstractions and the packages that manage the components.

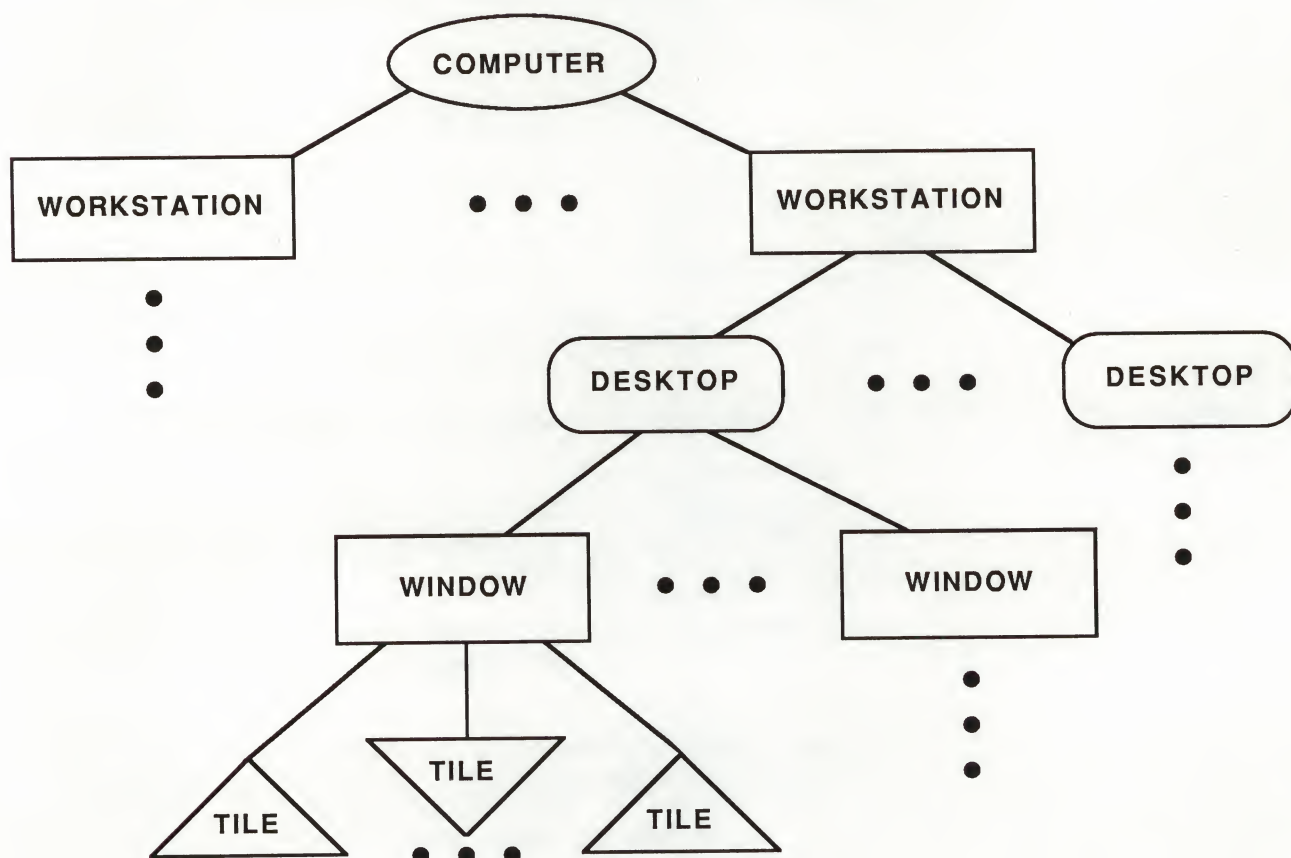
There is a hierarchy of abstractions that make up the window system:

- *Tiles* are used to tile the surface of a window. Tiles don't overlap and may not be nested. For example, a text subwindow with a scrollbar is implemented with separate tiles for both the scrollbar and the text portion of the subwindow.
- Windows are allowed to overlap one another¹ and may be arbitrarily nested. Frames, panels, text subwindows, canvases and the root window are all implemented as windows.
- Screens, sometimes called *desktops*, support multiple windows and represent physical display devices. A screen is covered by the root window.
- *Workstations* support multiple screens that share common user input devices on the behalf a single user. For example, one can slide the cursor between screens.

The figure below shows the hierarchy:

¹ The procedure which lays out subwindows of tools does it so they do not overlap, but this is not an inherent restriction.

Figure 3-1 SunView System Hierarchy



Data Managers

The various parts of the system support the management of this hierarchy. They provide the glue between the various components:

- The *window driver*, (currently) residing in the UNIX kernel as a pseudo device driver that is accessed through library routines, supports windows, screens and workstations.
- The *pixwin* library package allows implementors of specific windows and tiles to access the screen for drawing.
- The Notifier library package is used to support the general flow of control to multiple disjoint clients.
- The Agent library package can be viewed as the SunView-specific extension of the Notifier. The Agent supports tiles and windows.
- The Selection Service is a separate user process that supports the inter-process communication and control of user selection related data. In this role it essentially supports specific tile implementations.

Data Representations

This conceptual model is useful to understand the structure and workings of the system. However, the model doesn't always translate into corresponding objects:

- Tiles are implemented as opaque handles with *pixwin* regions used to communicate the size and position of the tile to the Agent.
- At the system level, windows are implemented as UNIX *devices* which are represented by *file descriptors*. Window devices are not to be confused with the application level notion of windows which are opaque handles. A *file descriptor* is returned by `open(2)` of an entry in the `/dev` directory. It is manipulated by other system calls, such as `select(2)`, `read(2)`, `ioctl(2)`, and `close(2)`.
- There is a screen structure that describes a limited number of properties of a desktop. However, it is a window file descriptor that is used as the "ticket" into the window driver to get and set screen related data. This is possible because a window is directly associated with a particular screen.
- There is no system object that translates into a workstation. However, like desktop data, workstation-related data is accessed using a window file descriptor. Again, this is because a window is directly associated with a particular screen which is directly associated with a particular workstation. As a side effect of this association, one can use the file descriptor of a panel and asked about workstation related data for the workstation on which the panel resides.

3.2. Model Dynamics

Now that you have been introduced to the players in the window system, let's see how they interact.

Tiles and the Agent

Tiles are quite simple and “lightweight” abstractions. The main reason for having tiles instead of yet another nesting of windows is that file descriptors are relatively heavyweight. There can only be 64 file descriptors open per UNIX process in Sun’s release 4.0. As a result, a tile provides only a subset of the functionality of a full-blown window. After telling the Agent that a tile covers a certain portion of the window, the Agent provides the following services:

- The Agent tells you when your tile has been resized.
- The Agent tells you when your tile should be repainted. Optionally, you can tell the Agent to maintain a retained image for your tile from which the Agent can handle the repainting itself.
- The Agent reads input for the tile’s window and distributes it to the appropriate tile.
- The Agent notices when tile regions have been entered and exited by the cursor and notifies the tile.

In addition, the Agent is the conduit by which client generated events are passed between tiles. For example, when the scrollbar wants to tell a canvas that it should now scroll, the communications is arranged via the Agent. The Agent, in turn, uses the Notifier to implement the data transfer.

It is your responsibility to lay out your window’s tiles so that they don’t overlap, even when the window size changes.

Even a window with only a single tile that covers its entire surface may use the Agent and its features.

Windows

Windows are the focus of most of the functionality of the window system. Here is a list of the information about a window maintained by the window system:

- A rectangle refers to the *size* and *position* of a window. Some windows (frames) also utilize an alternative rectangle that describes the iconic position of a window.
- Each window has a series of links that describe the window’s position in a hierarchical database, which determines its *overlapping* relationships to other windows. Windows may be arbitrarily nested, providing distinct *subwindows* within an application’s screen space.
- Arbitration between windows is provided in the allocation of display space. Where one window limits the space available to another, *clipping*, guarantees that one does not interfere with the other’s image. One such conflict arises when windows share the same coordinates on the display: one overlaps the other. Thus, clipping information is associated with each window.
- When one window impacts another window’s image without any action on the second window’s part, the window system informs the affected window of the damage it has suffered, and the areas that ought to be repaired. To do this the window system maintains a description of the portion of the window of the display that is corrupted as well as the process id of the window’s owner.

- On color displays, colormap entries are a scarce resource. When shared among multiple applications, they become even more scarce: there may be simultaneous demand for more colors than the display can support. Arbitration between windows is provided in the allocation of colormap entries. Provisions are made to share portions of the colormap (*colormap segments*). There is colormap information that describes that portion of the colormap assigned to a window.
- Real-time response is important when tracking the cursor, so this is done by the window system. Thus, the image (cursor and optional cross hairs) used to track the mouse when it is in the window is part of the window's data.²
- Windows may be selective about which input events they will process, and rejected events will be offered to other windows for processing; you can explicitly designate the window rejected events are first offered to.³ A mask indicates what keyboard input actions the window should be notified of and there is a similar mask for pick/locator-related actions.
- A window device is read in order to receive the user input events directed at it. So like other input devices a window supports a variety of the input modes, such as blocking or non-blocking, synchronous or asynchronous, etc. In addition, there is a queue of input events that are pending for a window.
- There are 32 bits of data private to the window client stored with the window.

Desktop

Desktop data relates to the physical display:

- The physical display is associated with a UNIX device. The desktop maintains the name of this device.
- The desktop maintains the notion of a default foreground and background color.
- The desktop records the size of the screen.
- The desktop maintains the name of the distinguished root window on itself.
- When multiple screens are part of a workstation, each desktop knows the relative physical placement of its neighboring displays so that the mouse cursor may slide between them.

² There is only one cursor per window, but the image may be different in different tiles within the window (e.g. scrollbars have different cursors). If so, the different cursor images are dynamically loaded by the user process and thus real time response is not assured.

³ Not all events are passed on to a designee, for example window-specific events such as `LOC_WINENTER` and `KBD_REQUEST` are not.

Locking

The desktop also arbitrates screen surface access and window database manipulation.

Display Locking

Display locking prevents window processes from interfering with each other in several ways:

- Raster hardware may require several operations to complete a change to the display; one process' use of the hardware is protected from interference by others during this critical interval.
- Changes to the arrangement of windows must be prevented while a process is painting, lest an area be removed from a window as it is being painted.
- A software cursor that the window process does not control (the kernel is usually responsible for the cursor) may have to be removed so that it does not interfere with the window's image.

Window Database Locking

Window database locking is used when doing multiple changes to the window's size, position, or links in the window hierarchy. This prevents any other process from performing a conflicting modification and allows the window system to treat changes as atomic.

Colormap Sharing

On color displays, colormap entries are a limited resource. When shared among multiple applications, colormap usage requires arbitration. Consider the following applications running on the same display at the same time in different windows:

- Application program X needs 64 colors for rendering VLSI images.
- Application program Y needs 32 shades of gray for rendering black and white photographs.
- Application program Z needs 256 colors (assume this is the entire colormap) for rendering full color photographs.

Colormap usage control is handled as follows:

- To determine how X and Y figure out what portion of the colormap they should use (so they don't access each others' entries), the window system provides a resource manager that allocates a *colormap segment* to each window from the *shared colormap*. To reduce duplicate colormap segments, they are named and can be shared among cooperating processes.
- To hide concerns about the correct offset to the start of a colormap segment from routines that access the image, the window system initializes the image of a window with the colormap segment offset. This effectively hides the offset from the application.
- To accommodate Z if its large colormap segment request cannot be granted, Z's colormap is loaded into the hardware, replacing the shared colormap, whenever the cursor is over Z's window. Z's request is not denied even though it is not allocated its own segment in the shared colormap.

Workstations

The domain of a workstation is to manage the global state of input processing. User inputs are unified into a single stream within the window system, so that actions with the user input devices, usually a mouse and a keyboard, can be coordinated. This unified stream is then distributed to different windows, according to user or programmatic indications. To this end a workstation manages the following:

- A workstation needs some number of user input devices to run. A distinguished keyboard device and a distinguished mouse-like device are recognized since these are required for a useful workstation. Non-Sun supported user input devices may be used as these distinguished devices.
- Additional, non-distinguished user input devices, may be managed by a workstation as well.
- The input devices associated with the workstation are polled by the window system. Locator motion causes the cursor to move on the screen. Certain interrupt event sequences are noted. Events are timestamped enqueued on the workstation's input queue based on the time they were generated.
- This input queue is massaged in a variety of ways. If the input queue becomes full, locator motion events on the queue are compressed in order to reduce its size. In addition, locator motion at the head of the queue is (conditionally) collapsed so as to deliver the most up-to-date locator position to applications.
- Based on the state of input focuses and window input masks a window is selected to receive the next event from the head of the input queue. The event is placed on the window device's separate input pending queue and the window's process is awoken.
- The workstation uses a synchronized input mechanism. The main benefit of a synchronized input mechanism is that it removes the input race conditions inherent in a multiple process environment. While a window processes the input event the workstation waits for it to finish before handing out the next event.
- The workstation deals with situations in which a process takes too long to finish processing an input event by pressing on ahead in a partially synchronized mode until the errant process catches up to the user. This prevents a misbehaving process from disabling user interaction with other processes.

The Agent & Tiles

The Agent & Tiles	21
4.1. Registering a Tile With the Agent	21
Laying Out Tiles	22
Dynamically Changing Tile Flags	23
Extracting Tile Data	23
4.2. Notifications From the Agent	23
4.3. Posting Notifications Through the Agent	24
4.4. Removing a Tile From the Agent	26



The Agent & Tiles

This chapter describes how to utilize the Agent to manage tiles for you. It contains the implementation details associated with tiles and the Agent, as introduced in the *SunView System Model* chapter. This chapter uses a text subwindow with a scrollbar as an example of Agent utilization.⁴

4.1. Registering a Tile With the Agent

The Agent is a little funny in that you don't ask it to create a tile for you that it will then manage. In fact tiles are only abstractions. Instead, you create a *pixwin* region and a unique client object and pass these to the Agent to manage on your behalf. The following routine is how this registration is done.

```
int
win_register(client, pw, event_func, destroy_func, flags)
    Notify_client  client;
    Pixwin        *pw;
    Notify_func    event_func;
    Notify_func    destroy_func;
    u_int         flags;

#define PW_RETAIN          0x1
#define PW_FIXED_IMAGE    0x2
#define PW_INPUT_DEFAULT  0x4
#define PW_NO_LOC_ADJUST  0x8
#define PW_REPAINT_ALL    0x10
```

client is the handle that the Agent will hand back to you when you are notified of interesting events (see below) by a call to the *event_func* function. *client* is usually the same client handle by which a tile is known to the Notifier. Client handles need to be unique among all the clients registered with the Notifier.

pw is a *pixwin* opened by *client* and is the *pixwin* by which the tile writes to the screen. This *pixwin* could have been created by a call to *pw_open()* if the window has only a single tile that covers its entire surface. More often the tile covers a region of the windows created by a call to *pw_region()*, documented in the *Clipping with Regions* section of the *Imaging Facilities: Pixwins* chapter

⁴ The header file */usr/include/sunwindow/window_hs.h* contains the definitions for the routines in this chapter.

of the *SunView 1 Programmer's Guide*. Regions are themselves pixwins that refer to an area within an existing pixwin.

flags control the options utilized by the Agent when managing your tile:

- PW_RETAIN — Your tile will be managed as retained. This means that the window system maintains a backup image of your tile in memory from which the screen can be refreshed in case the tile is exposed after being hidden.
- PW_FIXED_IMAGE — The underlaying abstraction of the image that your tile is displaying is fixed in size. This means that the client need not be asked to repaint the entire tile on a window size change. Only the newly exposed parts need be repainted.
- PW_INPUT_DEFAULT — Usually, the cursor position over a tile indicates which tile input will be sent to. However, if your window has the keyboard focus, the cursor need not be over any tile in your window in order for the window to be sent input. The tile with this flag on will receive input if the cursor is not over any tile in the window. In our example, the text display tile would be created with this flag on because it is the main tile in the window.
- PW_NO_LOC_ADJUST — Usually, when the Agent notifies your tile of an event the locator x and y positions contained in your event are adjusted to be relative to the tile's upper left hand corner. Turning this flag on suppresses this action which means that you'll get events in the window's coordinate space.
- PW_REPAINT_ALL — Setting this flag causes your tile to be completely repainted when ever the Agent detects that any part of your window needs to be repainted.

event_func is the client event notification function for the tile and destroy_func is the client destroy function for the tile. The Agent actually sets these functions up with the notifier (see the *Notifier* chapter in the *SunView 1 Programmer's Guide* for a discussion of these two types of notification functions and their calling conventions). In addition, the Agent gets input pending and SIGWINCH received (used for repaint and resize detection) notifications from the notifier and posts corresponding events to the appropriate tile. Tiles in the same window need to share the same input pending notification procedure because input is distributed from the kernel at a window granularity. Tiles also share the same input masks, as well as other window data.

Laying Out Tiles

Tiles are used to tile the surface of a window. Tiles may not overlap and may not be nested. As an example, a text subwindow with a scrollbar is implemented with a separate tile for both the scrollbar and the text portion of the subwindow. It is a window owner's responsibility to layout tiles so that they don't overlap. The Agent does nothing for you in this regard, so layout is arranged via conventions among tiles. In our example, there are two tiles, the scrollbar and a text display area. Here is how layout works when scrollbars are involved:

- The text subwindow code creates a vertical scrollbar. The scrollbar code looks at the user's scrollbar defaults and finds out what side to put the scrollbar on and how wide it should be. Given this information it figures out where to place its tile. The scrollbar code registers its new tile with the Agent.
- After creating the scrollbar, the text subwindow code asks the scrollbar what side it is on and how thick it is. Given this information the text subwindow figures out where to place its text display tile. The text subwindow code registers its new tile with the Agent.
- When a window resize notification (sent by the Agent) is received by the scrollbar it knows to hug the side that it is on as it adjusts the size of its region. A similar arrangement is followed by the text display tile.

Dynamically Changing Tile Flags

The following routine lets you dynamically set the tile's flags:

```
int
win_set_flags(client, flags)
    Notify_client  client;
    u_int          flags;
```

A -1 is returned if `client` is not registered, otherwise 0 is returned.

When you set a single flag, it is best to retrieve the state of all the flags first and then operate on the bit that you are changing, then write all the flags back; otherwise, any other flags that are set will be reset. The following routine retrieves the current flags of the tile associated with `client`:

```
u_int
win_get_flags(client)
    Notify_client  client;
```

Extracting Tile Data

Extraction of interesting values from clients of the Agent is done via the following calls:

```
int
win_get_fd(client)
    Notify_client  client;
```

`win_get_fd()` gets the window file descriptor associated with `client`'s tile.

```
Pixwin *
win_get_pixwin(client)
    Notify_client  client;
```

`win_get_pixwin()` gets the `pixwin` associated with `client`'s tile.

4.2. Notifications From the Agent

Once you register your tile with the Agent, the Agent causes the `event_func` you passed to `win_register()` to be called ("notified") to handle events. You must write your tile's event notification procedure yourself; the events it might receive are listed in the *Handling Input* chapter in the *SunView 1 Programmer's Guide*.

The calling sequence for any client event notification function is:

```
Notify_value
event_func(client, event, arg, when)
    Notify_client    client;
    Event            *event;
    Notify_arg       arg;
    Notify_event_type when;
```

`client` is the client handle passed into `win_register()`. `event` is the event your tile is notified of. `arg` is usually `NULL` but depends on `event_id(event)`. In the case of the scrollbar tile notifying the text display tile of a scroll action `arg` is actually defined. `when` is described in the chapter *Advanced Notifier Usage* and is usually `NOTIFY_SAFE`.

What your tile does with events is largely up to you; however, there are a few things to note about certain classes of events.

- For `LOC_RGNENTER` and `LOC_RGNEXIT` to be generated for tiles, `LOC_MOVE`, `LOC_WINENTER` and `LOC_WINEXIT` need to be turned on. Remember that tiles share their window's input mask so they need to cooperate in their use of it.
- Locator coordinate translation is done so that the event is relative to a tile's coordinate system unless disabled by `PW_NO_LOC_ADJUST`.
- On a `WIN_RESIZE` event, you can use `pw_set_region_rect()` to change the size and position of your tile's pixwin region.
- On a `WIN_REPAINT`, you simply repaint your entire tile. The Agent will have set the clipping of your pixwin so that only the minimum portion of the screen will actually appear to repaint. Alternatively, if you have initially told the Agent to maintain a retained image for your tile from which the Agent can handle the repainting itself, you will only get a `WIN_REPAINT` call after a window size change. You won't even get this call if your tile's flags have `PW_FIXED_IMAGE` and `PW_RETAIN` bits turned on.

4.3. Posting Notifications Through the Agent

The Agent is the conduit by which client-generated events are passed between tiles. For example, when the scrollbar wants to tell a canvas that it should now scroll, the communications is arranged via the Agent. The Agent, in turn, uses the Notifier to implement the data transfer.

The Agent follows the lead of the Notifier when it comes to posting events. See the documentation on `notify_post_event()` and `notify_post_event_and_arg()` in the *Advanced Notifier Usage* chapter if you are going to be posting events between tiles.

There are four routines available for posting an event to another tile.

```
Notify_error
win_post_id(client, id, when_hint)
    Notify_client    client;
    short            id;
    Notify_event_type when_hint;
```


is provided if you want to send an event to a tile and you don't really care about any event data except the `event_id(event)`. The Agent will generate the remainder of the event for you with up-to-date data. `when_hint` is usually `NOTIFY_SAFE`.

A second routine is available if you want to manufacture an event yourself. This is easy if you already have an event in hand.

```
Notify_error
win_post_event(client, event, when_hint)
    Notify_client    client;
    Event            *event;
    Notify_event_type when_hint;
```

The other two routines parallel the first two but include the capability to pass an arbitrary additional argument to the destination tile. The calling sequence is more complicated because one must make provisions to copy and later free the additional argument in case the delivery of the event is delayed.

```
Notify_error
win_post_id_and_arg(client, id, when_hint, arg,
                    copy_func, release_func)
    Notify_client    client;
    short            id;
    Notify_event_type when_hint;
    Notify_arg        arg;
    Notify_copy        copy_func;
    Notify_release     release_func;
```

```
Notify_error
win_post_event_arg(client, event, when_hint, arg,
                   copy_func, release_func)
    Notify_client    client;
    Event            *event;
    Notify_event_type when_hint;
    Notify_arg        arg;
    Notify_copy        copy_func;
    Notify_release     release_func;
```

The copy and release functions are covered in the *Advanced Notifier Usage* chapter. After reading about them you will know why you need the following utilities to copy the event as well as the arg:

```
Notify_arg
win_copy_event(client, arg, event_ptr)
    Notify_client    client;
    Notify_arg        arg;
    Event            **event_ptr;

void
win_free_event(client, arg, event)
    Notify_client    client;
    Notify_arg        arg;
    Event            *event;
```


4.4. Removing a Tile From the Agent

The following call tells the Agent to stop managing the tile associated with `client`.

```
int  
win_unregister(client)  
    Notify_client client;
```

You should call this from the tile's `destroy_func` that you gave to the Agent in the `win_register()` call. `win_unregister()` also completely removes `client` from having any conditions registered with the Notifier. A `-1` is returned if `client` is not registered, otherwise 0 is returned.

Windows

Windows	29
5.1. Window Creation, Destruction, and Reference	29
A New Window	29
An Existing Window	30
References to Windows	30
5.2. Window Geometry	31
Querying Dimensions	31
The Saved Rect	32
5.3. The Window Hierarchy	32
Setting Window Links	32
Activating the Window	33
Defaults	33
Modifying Window Relationships	34
Window Enumeration	35
Enumerating Window Offspring	35
Fast Enumeration of the Window Tree	36
5.4. Pixwin Creation and Destruction	36
Creation	36
Region	37
Retained Image	37
Bell	37
Destruction	37
5.5. Choosing Input	37

Input Mask	37
Manipulating the Mask Contents	38
Setting a Mask	38
Querying a Mask	39
The Designee	39
5.6. Reading Input	39
Non-blocking Input	39
Asynchronous Input	39
Events Pending	40
5.7. User Data	40
5.8. Mouse Position	40
5.9. Providing for Naive Programs	41
Which Window to Use	41
The Blanket Window	41
5.10. Window Ownership	42
5.11. Environment Parameters	42
5.12. Error Handling	43

Windows

This chapter describes the facilities for creating, positioning, and controlling windows. It contains the implementation details associated with window devices, as introduced in the *SunView System Model* chapter.

NOTE *The recommended window programming approach is described in the SunView 1 Programmer's Guide. You should only resort to the following window device routines if the equivalent isn't available at the higher level. It is possible to use the following routines with a high level SunView Window object by passing the file descriptor returned by*

```
(int) window_get(Window_object, WIN_FD);
```

The structure that underlies the operations described in this chapter is maintained within the window system, and is accessible to the client only through system calls and their procedural envelopes; it will not be described here. The window is presented to the client as a *device*; it is represented, like other devices, by a *file descriptor* returned by `open(2)`. It is manipulated by other UNIX system calls, such as `select(2)`, `read(2)`, `ioctl(2)`, and `close(2)`.⁵

5.1. Window Creation, Destruction, and Reference

As mentioned above, windows are *devices*. As such, they are special files in the `/dev` directory with names of the form `"/dev/winn,"` where *n* is a decimal number. A window is created by opening one of these devices, and the window name is simply the filename of the opened device.

A New Window

The first process to open a window becomes its *owner*. A process can obtain a window it is guaranteed to own by calling:

```
int
win_getnewwindow()
```

This finds the first unopened window, opens it, and returns a file descriptor which refers to it. If none can be found, it returns `-1`. A file descriptor, often called the *windowfd*, is the usual handle for a window within the process that opened it.

When a process is finished with a window, it may close it with the standard `close(2)` system call with the window's file descriptor as its argument. As with other file descriptors, a window left open when its owning process terminates

⁵ The header file `<sunwindow/window_hs.h>` includes the header files needed to work at this level of the window system. The library `/usr/lib/libsunwindow.a` implements window device routines.

will be closed automatically by the operating system.

Another procedure is most appropriately described at this point, although in fact clients will have little use for it. To find the next available window, `win_getnewwindow()` uses:

```
int
win_nextfree(fd)
    int fd;
```

where `fd` is any valid window file descriptor. The return value is a *window number*, as described in *References to Windows* below; a return value of `WIN_NULLLINK` indicates there is no available unopened window.

An Existing Window

It is possible for more than one process to have a window open at the same time; the section *Providing for Naive Programs* below presents one plausible scenario for using this capability. The window will remain open until all processes which opened it have closed it. The coordination required when several processes have the same window open is described in *Providing for Naive Programs*.

References to Windows

Within the process which created a window, the usual handle on that window is the file descriptor returned by `open(2)` or `win_getnewwindow()`. Outside that process, the file descriptor is not valid; one of two other forms must be used. One form is the *window name* (e.g. `/dev/win12`); the other form is the *window number*, which corresponds to the numeric component of the window name. Both of these references are valid across process boundaries. The *window number* will appear in several contexts below.

Procedures are supplied for converting among various window identifiers. `win_numbertoname()` stores the filename for the window whose number is `winnumber` into the buffer addressed by `name`:

```
win_numbertoname(winnumber, name)
    int winnumber;
    char *name;
```

`name` should be `WIN_NAMESIZE` long, as should all the name buffers in this section.

`win_nametonenumber()` returns the window number of the window whose `name` is passed in `name`:

```
int
win_nametonenumber(name)
    char *name;
```

Given a window file descriptor, `win_fdtoname()` stores the corresponding device name into the buffer addressed by `name`:

```
win_fdtoname(windowfd, name)
    int windowfd;
    char *name;
```


`win_fdtonumber()` returns the window number for the window whose file descriptor is `windowfd`:

```
int
win_fdtonumber(windowfd)
    int windowfd;
```

5.2. Window Geometry

Once a window has been opened, its size and position may be set. The same routines used for this purpose are also helpful for adjusting the screen positions of a window at other times, when the window is to be moved or stretched, for instance. `win_setrect()` copies the `rect` argument into the rectangle of the indicated window:

```
win_setrect(windowfd, rect)
    int    windowfd;
    Rect *rect;
```

This changes its size and/or position on the screen. The coordinates in the `rect` structure are in the coordinate system of the window's parent. The *Rects and Rectlists* chapter explains what is meant by a `rect`. *Setting Window Links* below explains what is meant by a window's "parent." Changing the size of a window that is visible on the screen or changing the window's position so that more of the window is now exposed causes a chain of events which redraws the window. See the section entitled *Damage* in the *Advanced Imaging* chapter.

Querying Dimensions

The window size querying procedures are:

```
win_getrect(windowfd, rect)
    int    windowfd;
    Rect *rect;

win_getsize(windowfd, rect)
    int    windowfd;
    Rect *rect;

short win_getheight(windowfd)
    int    windowfd;

short win_getwidth(windowfd)
    int    windowfd;
```

`win_getrect()` stores the rectangle of the window whose file descriptor is `windowfd` into the `rect`; the origin is relative to that window's parent.

`win_getsize()` is similar, but the rectangle is self-relative — that is, the origin is (0,0).

`win_getheight()` and `win_getwidth()` return the single requested dimension for the indicated window — these are part of the `rect` structure that the other calls return.

The Saved Rect

A window may have an alternate size and location; this facility is useful for storing a window's iconic position that is associated with frames. The alternate rectangle may be read with `win_getsavedrect()`, and written with `win_setsavedrect()`.

```
win_getsavedrect(windowfd, rect)
    int    windowfd;
    Rect *rect;

win_setsavedrect(windowfd, rect)
    int    windowfd;
    Rect *rect;
```

As with `win_getrect()` and `win_setrect()`, the coordinates are relative to the screen.

5.3. The Window Hierarchy

Position in the window database determines the nesting relationships of windows, and therefore their overlapping and obscuring relationships. Once a window has been opened and its size set, the next step in creating a window is to define its relationship to the other windows in the system. This is done by setting links to its neighbors, and inserting it into the window database.

Setting Window Links

The window database is a strict hierarchy. Every window (except the root) has a parent; it also has 0 or more *siblings* and *children*. In the terminology of a family tree, *age* corresponds to *depth* in the layering of windows on the screen: parents underlie their offspring, and older windows underlie younger siblings which intersect them on the display. Parents also enclose their children, which means that any portion of a child's image that is not within its parent's rectangle is clipped. Depth determines overlapping behavior: the *uppermost* image for any point on the screen is the one that gets displayed. Every window has links to its parent, its older and younger siblings, and to its oldest and youngest children.

Windows may exist outside the structure which is being displayed on a screen; they are in this state as they are being set up, for instance.

The links from a window to its neighbors are identified by *link selectors*; the value of a link is a *window number*. An appropriate analogy is to consider the *link selector* as an array index, and the associated *window number* as the value of the indexed element. To accommodate different viewpoints on the structure there are two sets of equivalent selectors defined for the links:

```
WL_PARENT      == WL_ENCLOSING
WL_OLDER_SIB   == WL_COVERED
WL_YOUNGER_SIB == WL_COVERING
WL_OLDESTCHILD == WL_BOTTOMCHILD
WL_YOUNGESTCHILD == WL_TOPCHILD
```

A link which has no corresponding window, for example, a child link of a "leaf" window, has the value `WIN_NULLLINK`.

When a window is first created, all its links are null. Before it can be used for anything, at least the parent link must be set so that other routines know with which desktop and workstation this window is to be associated. If the window is

to be attached to any siblings, those links should be set in the window as well. The individual links of a window may be inspected and changed by the following procedures.

`win_getlink()` returns a window number.

```
int
win_getlink(windowfd, link_selector)
    int windowfd, link_selector;
```

This number is the value of the selected link for the window associated with `windowfd`.

```
win_setlink(windowfd, link_selector, value)
    int windowfd, link_selector, value;
```

`win_setlink()` sets the selected link in the indicated window to be `value`, which should be another window number or `WIN_NULLLINK`. The actual window number to be supplied may come from one of several sources. If the window is one of a related group, all created in the same process, file descriptors will be available for the other windows. Their window numbers may be derived from the file descriptors via `win_fdtonumber()`. The window number for the parent of a new window or group of windows is not immediately obvious, however. The solution is a convention that the `WINDOW_PARENT` environment parameter will be set to the filename of the parent. See `we_setparentwindow()` for a description of this parameter.

Activating the Window

Once a window's links have all been defined, the window is inserted into the tree of windows and attached to its neighbors by a call to

```
win_insert(windowfd)
    int windowfd;
```

This call causes the window to be inserted into the tree, and all its neighbors to be modified to point to it. This is the point at which the window becomes available for display on the screen.

Every window should be inserted after its rectangle(s) and link structure have been set, but the insertion need not be immediate: if a subtree of windows is being defined, it is appropriate to create the window at the root of this subtree, create and insert all of its descendants, and then, when the subtree is fully defined, insert its root window. This activates the whole subtree in a single action, which may result in cleaner display of the whole tree.

Defaults

One need not specify all the sibling links of a window that is being inserted into the display tree. Sibling links may be defaulted as follows (these conventions are applied in order):

- If the `WL_COVERING` sibling link is `WIN_NULLLINK` then the window is put on the top of the heap of windows.

- If the WL_COVERED sibling link is WIN_NULLLINK then the window is put on the bottom of the heap of windows.
- If the WL_COVERED or WL_COVERING sibling links are invalid then the window is put on the bottom of the heap of windows.

Once a window has been inserted in the window database, it is available for input and output. At this point, it is appropriate to access the screen with pixwin calls (to draw something in the window!).

Modifying Window Relationships

Windows may be rearranged in the tree. This will change their overlapping relationships. For instance, to bring a window to the top of the heap, it should be moved to the "youngest" position among its siblings. And to guarantee that it is at the top of the display heap, each of its ancestors must likewise be the youngest child of *its* parent.

To accomplish such a modification, the window should first be removed:

```
win_remove(windowfd)
    int windowfd;
```

After the window has been removed from the tree, it is safe to modify its links, and then reinsert it.

A process doing multiple window tree modifications should lock the window tree before it begins. This prevents any other process from performing a conflicting modification. This is done with a call to:

```
win_lockdata(windowfd)
    int windowfd;
```

After all the modifications have been made and the windows reinserted, the lock is released with a call to:

```
win_unlockdata(windowfd)
    int windowfd;
```

Nested pairs of calls to lock and unlock the window tree are permitted. The final unlock call actually releases the lock.

If a client program uses any of the window manager routines, use of win_lockdata() and win_unlockdata() is not necessary. See the chapter on *Window Management* for more details.

Most routines described in this chapter, including the four above, will block temporarily if another process either has the database locked, or is writing to the screen, and the window adjustment has the possibility of conflicting with the window that is being written.

As a method of deadlock resolution, SIGXCPU is sent to a process that spends more than 2 seconds of process virtual time inside a window data lock, and the lock is broken.⁶

⁶ The section *Kernel Tuning Options* in the *Workstation* chapter describes how to modify this default number of seconds (see ws_lock_limit).

Window Enumeration

There are routines that pass a client-defined procedure to a subset of the tree of windows, and another that returns information about an entire layer of the window tree. They are useful in performing window management operations on groups of windows. The routines and the structures they use and return are listed in `<sunwindow/win_enum.h>`.

Enumerating Window Offspring

The routines `win_enumerate_children()` and `win_enumerate_subtree()` repeatedly call the client's procedure passing it the windowfds of the offspring of the client window, one after another:

```
enum win_enumerator_result
win_enumerate_children(windowfd, proc, args);
    Window_handle  windowfd;
    Enumerator      proc;
    caddr_t         args;

enum win_enumerator_result
win_enumerate_subtree(windowfd, proc, args);
    Window_handle  windowfd;
    Enumerator      proc;
    caddr_t         args;

enum win_enumerator_result \
{ Enum_Normal, Enum_Succeed, Enum_Fail };

typedef enum win_enumerator_result
    (*Enumerator)();
```

`windowfd` is the window whose children are enumerated (`Window_handle` is typedef'd to `int`). Both routines repeatedly call `proc()`, stopping when told to by `proc()` or when everything has been enumerated.

`proc()` is passed a `windowfd` and `args`

```
(*proc)(fd, args);
```

It does whatever it wants with the `windowfd`, then returns `win_enumerator_result`. If `proc()` returns `Enum_Normal` then the enumeration continues; if it returns `Enum_Succeed` or `Enum_Fail` then the enumeration halts, and `win_enumerate_children` or `win_enumerate_subtree()` returns the same result.

The difference between the two enumeration procedures is that `win_enumerate_children()` invokes `proc()` with an `fd` for each *immediate* descendant of `windowfd` in oldest-to-youngest order, while `win_enumerate_subtree()` invokes `proc()` with a `windowfd` for the *original* window `windowfd` and for *all* of its descendants in depth-first, oldest-to-youngest order. The former enumerates `windowfd`'s children, the latter enumerates `windowfd` and its extended family.

It is possible that `win_enumerate_subtree()` can run out of file descriptors during its search of the tree if the descendants of `windowfd` are deeply nested.

Fast Enumeration of the Window Tree

The disadvantage with the above two routines is that they are quite slow. They traverse the window tree, calling `win_getlink()` to find the offspring, then open each window, then call the procedure.

In 3.2 there is a fast window routine, `win_get_tree_layer()`, that returns information about all the children of a window in a single ioctl⁷:

```
win_get_tree_layer(parent, size, buffer);
Window_handle parent;
u_int size;
char *buffer;

typedef struct win_enum_node {
    unsigned char me;
    unsigned char parent;
    unsigned char upper_sib;
    unsigned char lowest_kid;
    unsigned int flags;
#define WIN_NODE_INSERTED 0x1
#define WIN_NODE_OPEN 0x2
#define WIN_NODE_IS_ROOT 0x4
    Rect open_rect;
    Rect icon_rect;
} Win_enum_node;
```

`win_get_tree_layer()` fills `buffer` with `Win_enum_node` information (rects, user_flags, and minimal links) for the children of window in oldest-to-youngest order. It returns the number of bytes of `buffer` filled with information, or negative if there is an error.

Unlike `win_enumerate_children()`, `win_get_tree_layer()` returns information for all the children of `windowfd` including those that are have not been installed in the window tree with `win_insert()`; such children will not have the `WIN_NODE_INSERTED` flag set.

5.4. Pixwin Creation and Destruction

A pixwin is the object that you use to access the screen. Its usage has been covered in the *Imaging Facilities: Pixwins* chapter of the *SunView Programmer's Guide* and in the previous chapter on tiles. How to create a pixwin region has also been covered in the same places. Here we cover how a pixwin is generated for a window.

Creation

To create a pixwin, the window to which it will refer must already exist. This task is accomplished with procedures described earlier in this chapter. The pixwin is then created for that window by a call to `pw_open()`:

```
Pixwin *
pw_open(windowfd)
    int windowfd;
```

`pw_open()` takes a file descriptor for the window on which the pixwin is to

⁷ `win_get_tree_layer()` will use the slower method if the kernel does not support the ioctl; thus programs that use this can be run on 3.0 systems.

write. A pointer to a `pixwin` struct is returned. At this point the `pixwin` describes the exposed area of the window.

Region

To create the `pixwin` for a tile, call `pw_region()` passing it the `pixwin` returned from `pw_open()`.

Retained Image

If the client wants a *retained pixwin*, the `pw_prretained` field of the `pixwin` should be set to point to a memory `pixrect` of your own creation. If you set this field you need to call `pw_exposed(pw)` afterwards.⁸ This updates the `pixwin`'s exposed area list to deal with the memory `pixrect`; see the *Advanced Imaging* chapter for more information on `pw_expose()`.

Bell

The following routine can be used to beep the keyboard bell and flash a `pixwin`:

```
win_bell(windowfd, wait_tv, pw)
    int      windowfd;
    struct timeval wait_tv;
    Pixwin *pw;
```

If `pw` is 0 then there is no flash. `wait_tv` controls the duration of the bell.⁹

Destruction

When a client is finished with a `pixwin`, it should be released by a call to:

```
pw_close(pw)
    Pixwin *pw;
```

`pw_close()` frees any resource associated with the `pixwin`, including its `pw_prretained` `pixrect` if any. If the `pixwin` has a lock on the screen, it is released.

5.5. Choosing Input

The chapter entitled *Handling Input* in the *SunView Programmer's Guide* describes the window input mechanism. This section describes the file descriptor level interface to setting a window's input masks. This section is very terse, assuming that the concept from *Handling Input* are well understood.

Input Mask

Clients specify which input events they are prepared to process by setting the input masks for each window being read. The calls in this section manipulate input masks.

⁸ The best way to manage a retained window is to let the Agent do it (see `win_register()`).

⁹ The bell's behavior is controlled by the SunView `defaultsedit` entries *SunView/Audible_Bell* and *Sunview/Visible_Bell*, so the sound and flash can be disabled by the user, regardless of what the call to `win_bell()` specifies.


```
typedef struct inputmask {
    short    im_flags;
#define IM_NEGEVENT    (0x01) /* send input negative events too */
#define IM_ASCII      (0x10) /* enable ASCII codes 0-127 */
#define IM_META       (0x20) /* enable META codes 128-255 */
#define IM_NEGASCII   (0x40) /* enable negative ASCII codes 0-127 */
#define IM_NEGMETA    (0x80) /* enable negative META codes 128-255 */
#define IM_INTRANSIT  (0x400) /* don't suppress locator events when
                                in-transit over window */
    ...
} Inputmask;
```

Manipulating the Mask Contents

The bit flags defined for the input mask are stored directly in the `im_flags` field. To set a particular event in the input mask use the following macro:

```
win_setinputcodebit(im, code)
    Inputmask *im;
    u_short    code;
```

`win_setinputcodebit()` sets a bit indexed by `code` in the input mask addressed by `im` to 1.

```
win_unsetinputcodebit(im, code)
    Inputmask *im;
    u_short    code;
```

`win_unsetinputcodebit()` resets the bit to zero.

The following macro is used to query the state of an event code in an input mask:

```
int
win_getinputcodebit(im, code)
    Inputmask *im;
    u_short    code;
```

`win_getinputcodebit()` returns non-zero if the bit indexed by `code` in the input mask addressed by `im` is set.

`input_imnull()` initializes an input mask to all zeros:

```
input_imnull(im)
    Inputmask *im;
```

It is critical to initialize the input mask explicitly when the mask is defined as a local procedure variable.

Setting a Mask

The following routines set the keyboard and pick input masks for a window. The different types of masks are discussed in the *Input* chapter.

```
win_set_kbd_mask(windowfd, im)
    int        windowfd;
    Inputmask *im;

win_set_pick_mask(windowfd, im)
    int        windowfd;
    Inputmask *im;
```

Querying a Mask

The following routines get the keyboard and pick input masks for a window.

```

win_get_kbd_mask(windowfd, im)
    int      windowfd;
    Inputmask *im;

win_get_pick_mask(windowfd, im)
    int      windowfd;
    Inputmask *im;

```

The Designee

The designee is that window that input is directed to if the input mask for a window doesn't match a particular event:

```

win_get_designee(windowfd, nextwindownumber)
    int windowfd, *nextwindownumber;

win_set_designee(windowfd, nextwindownumber)
    int windowfd, nextwindownumber;

```

5.6. Reading Input

The recommended way of getting input is to let the Agent notify you of input events (see chapter on tiles). However, there are times when you may want to read input directly, say, when tracking the mouse until one of its buttons goes up. A library routine exists for reading the next input event for a window:

```

int
input_readevent(windowfd, event)
    int      windowfd;
    Event *event;

```

This fills in the event struct, and returns 0 if all went well. In case of error, it sets the global variable `errno`, and returns -1; the client should check for this case.

Non-blocking Input

A window can be set to do either blocking or non-blocking reads via a standard `fcntl(2)` system call, as described in `fcntl(2)` (using `_SETFL`) and `fcntl(5)` (using `FNDELAY`). A window defaults to blocking reads. The blocking status of a window can be determined by the `fcntl(2)` system call.

When all events have been read and the window is doing non-blocking I/O, `input_readevent()` returns -1 and the global variable `errno` is set to `EWOULDBLOCK`.

Asynchronous Input

A window process can ask to be sent a `SIGIO` if any input is pending in a window. This option is also enabled via a standard `fcntl(2)` system call, as described in `fcntl(2)` (using `F_SETFL`) and `fcntl(5)` (using `FASYNC`). The programmer can set up a signal handler for `SIGIO` by using the `notify_set_signal_func()` call.¹⁰

¹⁰ The Notifier handles asynchronous input without you having to set up your own signal handler if you are using the Notifier to determine when there is input for a window.

Events Pending

The number of character in the input queue of a window can be determined via a `FBIONREAD ioctl(2)` call. `FBIONREAD` is described in `tty(4)`. Note that the value returned is the number of bytes in the input queue. If you want the number of `Events` then divide by `sizeof(Event)`.

5.7. User Data

Each window has 32 bits of data associated with it. These bits are used to implement a minimal inter-process window-related status-sharing facility. Bits 0x01 through 0x08 are reserved for the basic window system; 0x01 is currently used to indicate if a window is a blanket window. Bits 0x10 through 0x80 are reserved for the user level window manager; 0x10 is currently used to indicate if a window is iconic. Bits 0x100 through 0x80000000 are available for the programmer's use. They is manipulated with the following procedures:

```
int
win_getuserflags(windowfd)
    int windowfd;

int
win_setuserflags(windowfd, flags)
    int windowfd;
    int flags;

int
win_setuserflag(windowfd, flag, value)
    int windowfd;
    int flag;
    int value;
```

`win_getuserflags()` returns the user data. `win_setuserflags()` stores its `flags` argument into the window struct. `win_setuserflag()` uses `flag` as a mask to select one or more flags in the data word, and sets the selected flags on or off as `value` is `TRUE` or `FALSE`.

5.8. Mouse Position

Determining the mouse's current position is treated in the *SunView Programmer's Guide*. The convention for a process tracking the mouse is to arrange to receive an input event every time the mouse moves; the mouse position is passed with *every* user input event a window receives.

The mouse position can be reset under program control; that is, the cursor can be moved on the screen, and the position that is given for the mouse in input events can be reset without the mouse being physically moved on the table top.

```
win_setmouseposition(windowfd, x, y)
    int windowfd, x, y;
```

`win_setmouseposition()` puts the mouse position at `(x, y)` in the coordinate system of the window indicated by `windowfd`. The result is a jump from the previous position to the new one without touching any points between. Input events occasioned by the move, such as window entry and exit and cursor changes, will be generated. This facility should be used with restraint, as many users are unhappy with a cursor that moves independent of their control.

Occasionally it is necessary to discover which window underlies the cursor, usually because a window is handling input for all its children. The procedure used

for this purpose is:

```
int
win_findintersect(windowfd, x, y)
    int windowfd, x, y;
```

where `windowfd` is the calling window's file descriptor, and `(x, y)` defines a screen position in that window's coordinate space. The returned value is a window number of a child of the calling window. If a child of the calling window doesn't fall under the given position `WIN_NULLLINK` is returned.

5.9. Providing for Naive Programs

There is a class of applications that are relatively unsophisticated about the window system, but want to run in windows anyway. For example, a graphics program may want a window in which to run, but doesn't want to know about all the details of creating and positioning it. This section describes a way of allowing for these applications.

Which Window to Use

The window system defines an important environment parameter, `WINDOW_GFX`. By convention, `WINDOW_GFX` is set to a string that is the device name of a window in which graphics programs should be run. This window should already be opened and installed in the window tree. Routines exist to read and write this parameter:

```
int
we_getgfxwindow(name)
    char *name

we_setgfxwindow(name)
    char *name
```

`we_getgfxwindow()` returns a non-zero value if it cannot find a value.

The Blanket Window

A good way to take over an existing window is to create a new window that becomes attached to and covers the existing window. Such a covering window is called a *blanket* window. The covered window will be called the *parent* window in this subsection because of its window tree relationship with a blanket window.¹¹

The appropriate way to make use of the blanket window facility is as follows: Using the parent window name from the environment parameter `WINDOW_GFX` (described above), `open(2)` the parent window. Get a new window to be used as the blanket window using `win_getnewwindow()`. Now call:

```
int
win_insertblanket(blanketfd, parentfd)
    int blanketfd, parentfd;
```

A zero return value indicates success. As the parent window changes size and position the blanket window will automatically cover the parent.

¹¹ It's a bad idea to take over an existing window using `win_setowner()`.

To remove the blanket window from on top of the parent window call:

```
win_removeblanket(blanketfd)
int blanketfd;
```

If the process that owns the window over which the blanket window resides dies before `win_removeblanket()` is called, the blanket window will automatically be removed and destroyed.

A non-zero return value from `win_isblanket()` indicates that `blanketfd` is indeed a blanket window.

```
int
win_isblanket(blanketfd)
int blanketfd;
```

5.10. Window Ownership

`SIGWINCH` signals are directed to the process that *owns* the window, the owner normally being the process that created the window. The following procedures read from and write to the window:¹² These routines are included for backwards compatibility.

```
int
win_getowner(windowfd)
int windowfd;

win_setowner(windowfd, pid)
int windowfd, pid;
```

`win_getowner()` returns the process id of the indicated window owner. If the owner doesn't exist, zero is returned. `win_setowner()` makes the process identified by `pid` the owner of the window indicated by `windowfd`. `win_setowner` causes a `SIGWINCH` to be sent to the new owner.

5.11. Environment Parameters

Environment parameters are used to pass well-established values to an application. They have the valuable property that they can communicate information across several layers of processes, not all of which have to be involved.

Every frame must be given the name of its *parent window*. A frame's parent window is the window in the display tree under which the frame window should be displayed. The environment parameter `WINDOW_PARENT` is set to a string that is the device name of the parent window. For a frame, this will usually be the name of the root window of the window system.

```
we_setparentwindow(windevname)
char *windevname;

sets WINDOW_PARENT to windevname.
```

¹² Do not use the two routines in this section for *temporarily* taking over another window.


```
int
we_getparentwindow(windevname)
    char *windevname;
```

gets the value of WINDOW_PARENT into windevname. The length of this string should be at least WIN_NAMESIZE characters long, a constant found in <sunwindow/win_struct.h>. A non-zero return value means that the WINDOW_PARENT parameter couldn't be found.

The environment parameter DEFAULT_FONT should contain the font file name used as the program's default (see pf_default()).

NOTE *This is retained for backwards compatibility. All programs set this variable, but only old-style SunWindows programs, gfx subwindow programs and raw pixwin programs use it to determine which font to use. SunView programs that don't set their own font use the SunView/Font defaults entry; you can use the **-Wt fontname** command line frame argument to change the font of SunView programs that allow it.*

5.12. Error Handling

Except as explicitly noted, the procedures described in this section do not return error codes. The standard error reporting mechanism inside the *sunwindow* library is to call an error handling routine that displays a message, typically identifying the ioctl(2) call that detected the error. This error message is somewhat cryptic; Appendix B, *Programming Notes*, has a section on *Error Message Decoding*. After the message display, the calling process resumes execution.

This default error handling routine may be replaced by calling:

```
int (*win_errorhandler(win_error)) ()
int (*win_error) ();
```

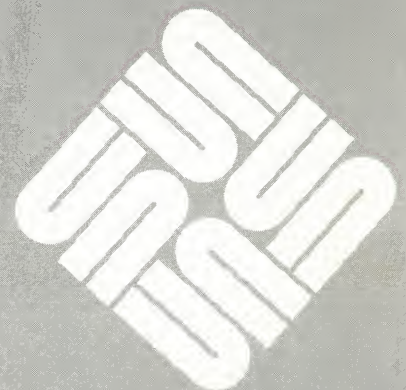
The win_errorhandler() procedure takes the address of one procedure, the new error handler, as an argument and returns the address of another procedure, the old error handler, as a result. Any error handler procedure should be a function that returns an integer.

```
win_error(errnum, winopnum)
    int errnum, winopnum;
```

errnum will be -1 indicating that the actual error number is found in the global errno. winopnum is the ioctl(2) number that defines the window operation that generated the error. See Section B.4, *Error Message Decoding*, in Appendix B, *Programming Notes*.

Desktops

Desktops	47
Look at sunview	47
6.1. Multiple Screens	47
The singlecolor Structure	47
The screen Structure	48
Screen Creation	48
Initializing the screen Structure	49
Screen Query	49
Screen Destruction	49
Screen Position	49
Accessing the Root FD	50



Desktops

This chapter discusses the calls that affect the screen, or desktop. Some calls affect workstation-related data, i.e., global input related data. This overlap of the conceptual model is purely historical.

Look at `sunview`

Many of the routines in here are used by Sun's window manager, `sunview(1)`. You will find it very helpful to look at the source for `sunview` (it is optional software that must be loaded in `suninstall`) to see how it uses these routines.

6.1. Multiple Screens

Workstations may use multiple displays, and clients may want windows on all of them.¹³ Therefore, the window database is a forest, with one tree of windows for each display. There is no overlapping of window trees that belong to different screens. For displays that share the same mouse device, the physical arrangement of the displays can be passed to the window system, and the mouse cursor will pass from one screen to the next as though they were continuous.

The `singlecolor` Structure

The screen structure describes attributes of the display screen. First, here is the definition of `singlecolor`, which it uses for the foreground and background colors:

```
struct singlecolor {  
    u_char  red, green, blue;  
};
```

¹³ There can be as many screens as there are frame buffers on your machine and *dtop* pseudo devices configured into your kernel. The kernel calls screen instances *dtops*.

The screen Structure

Now the screen structure:

```

struct screen {
    char      scr_rootname[SCR_NAMESIZE];
    char      scr_kbdname[SCR_NAMESIZE];
    char      scr_msname[SCR_NAMESIZE];
    char      scr_fbname[SCR_NAMESIZE];
    struct     singlecolor scr_foreground;
    struct     singlecolor scr_background;
    int        scr_flags;
    struct     rect scr_rect;
};

#define SCR_NAMESIZE 20
#define SCR_SWITCHBKGRDFRGRD 0x1

```

`scr_rootname` is the device name of the window which is at the base of the window display tree for the screen; it is often `/dev/win0`.¹⁴ `scr_kbdname` is the device name of the keyboard associated with the screen; the default is `/dev/kbd`. `scr_msname` is the device name of the mouse associated with the screen; the default is `/dev/mouse`. `scr_fbname` is the device name of the frame buffer on which the screen is displayed; the default is `/dev/fb` for the first desktop. `scr_kbdname`, `scr_msname` and `scr_fbname` can have the string "NONE" if no device of the corresponding type is to be associated with the screen. Workstations (hence also desktops) can have additional input devices associated with them; see the section on *User Input Device Control* in the *Workstations* chapter.

`scr_foreground` is three RGB color values that define the foreground color used on the frame buffer; the default is `{ colormap size-1, colormap size-1, colormap size-1 }`. `scr_background` is three RGB color values that define the background color used on the frame buffer; the default is `{ 0, 0, 0 }`. The default values of the background and foreground yield a black on white image. `scr_flags` contains boolean flags; the default is 0. `SCR_SWITCHBKGRDFRGRD` is a flag that directs any client of the background and foreground data to switch their positions, thus providing a video reversed image (usually yielding a white on black image). `scr_rect` is the size and position of the screen on the frame buffer; the default is the entire frame buffer surface.

Screen Creation

To create a new screen call:

```

int
win_screennew(screen)
    struct screen *screen;

```

¹⁴ Multiple screen configurations, in particular, will not have `/dev/win0` as the root window on the second screen.

`win_screennew()` opens and returns a window file descriptor for a root “desktop” window. This new root window resides on the new screen which was defined by the specifications of `screen`. Any zeroed field in `screen` tells `win_screennew()` to use the default value for that field (see above for defaults). Also, see the description of `win_initscreenfromargv()` below. If `-1` is returned, an error message is displayed to indicate that there was some problem creating the screen.

Initializing the screen Structure

The following routine can be called before calling `win_screennew()`:

```
int
win_initscreenfromargv(screen, argv)
    struct    screen *screen;
    char    **argv;
```

`win_initscreenfromargv()` zeroes the `*screen` structure, then it parses the relevant command line arguments in `argv` into `*screen`. You then call `win_screennew()` to create a root window with the desired attributes. The command line arguments allow the user to set all the variables in `*screen` including the display device, the keyboard device, the mouse device, the foreground and background colors, whether the screen colors should be inverted, and other features. See `sunview(1)` for semantics and details.

Screen Query

To find out about the screen on which your window is running call:

```
win_screenget(windowfd, screen)
    int            windowfd;
    struct screen *screen;
```

`win_screenget()` fills in the addressed struct `*screen` with information for the screen with which the window indicated by `windowfd` is associated. You can call this from any window.

Screen Destruction

To destroy the screen on which your window is running call:

```
win_screendestroy(windowfd)
    int windowfd;
```

`win_screendestroy()` causes each window owner process (except the invoking process) on the screen associated with `windowfd` to be sent a `SIGTERM` signal. This call will block until all the processes have died. If a window owner process hasn't gone away after 15 seconds, it is sent a `SIGKILL`, which will destroy it.

Screen Position

To tell the window system how multiple screens are arranged call:


```
win_setscreenpositions(windowfd, neighbors)
    int windowfd, neighbors[SCR_POSITIONS];

#define SCR_NORTH 0
#define SCR_EAST 1
#define SCR_SOUTH 2
#define SCR_WEST 3

#define SCR_POSITIONS 4
```

`win_setscreenpositions()` informs the window system of the logical layout of multiple screens. This enables the cursor to cross to the appropriate screen. `windowfd`'s window is the root for its screen; the four slots in `neighbors` should be filled in with the window numbers of the root windows for the screens in the corresponding positions. No diagonal neighbors are defined, since they are not strictly neighbors.

`win_getscreenpositions()` fills in `neighbors` with `windowfd`'s screen's neighbors:

```
win_getscreenpositions(windowfd, neighbors)
    int windowfd, neighbors[SCR_POSITIONS];
```

CAUTION In these routines, `windowfd` must be an fd for the root window. Most operations on the screen can be done using any `windowfd`.

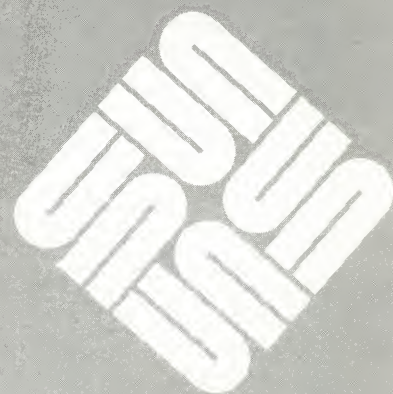
Accessing the Root FD

The following code fragment gets the screen struct for your window, then opens the window device of the root window:

```
int mywinfd,    rootfd;
struct screen  rootscreen;
...
win_screenget(mywinfd, &rootscreen);
rootfd = open(rootscreen.scr_rootname);
```

Workstations

Workstations	53
7.1. Virtual User Input Device	53
What Kind of Devices?	53
Vuid Features	54
Vuid Station Codes	54
Address Space Layout	54
Adding a New Segment	55
Input State Access	55
Unencoded Input	55
7.2. User Input Device Control	56
Distinguished Devices	56
Arbitrary Devices	57
Non-Vuid Devices	57
Device Removal	57
Device Query	57
Device Enumeration	58
7.3. Focus Control	58
Keyboard Focus Control	58
Event Specification	59
Setting the Caret Event	59
Getting the Caret Event	59
Restoring the Caret	59
7.4. Synchronization Control	60



Releasing the Current Event Lock	60
Current Event Lock Breaking	60
Getting/Setting the Event Lock Timeout	61
7.5. Kernel Tuning Options	61
Changing the User Actions that Affect Input	63

Workstations

This chapter discusses the manipulation of workstation data, which comprises mostly global data related to input and input devices. Some calls in the *Desktops* chapter also affect workstations. This overlap is purely historical. This chapter also explains parts of the Virtual User Input Device interface that were not covered in the *Handling Input* chapter of the *SunView 1 Programmer's Guide*. That chapter gave the possible event codes in SunView 1; this chapter explains the mechanism which sets up input devices to generate them.

7.1. Virtual User Input Device

The Virtual User Input Device (*vuid*) is an interface between input devices and their clients. The interface defines an idealized user input device that may not correspond to any existing physical collection of input devices. A client of SunView doesn't access *vuid* devices directly. Instead, the window system reads *vuid* devices, serializing input from all the *vuid* devices and then makes the input available to windows as SunView *vuid* events.

NOTE *You don't have to write a *vuid* interface to use your own device in SunView: you can use any input device that generates ASCII. But if your device is hooked up using *vuid*, then your SunView programs can interface with it using the SunView input event mechanism.*

Since SunView's input system is built on top of *vuid*, it is explained in some detail.

What Kind of Devices?

Vuid is targeted to input devices that gather command data from humans, e.g., mice, keyboards, tablets, joysticks, light pens, knobs, sliders, buttons, ascii terminals, etc.¹⁵ The *vuid* interface is not designed to support input devices that produce voluminous amounts of data, such as input scanners, disk drives, voice packets.

Here are some of the properties that are expected of a typical client of *vuid*, e.g., SunView:

- The client has a richer user interface than can be supported by a simple ASCII terminal.

¹⁵ The appendix titled *Writing a Virtual User Input Device Driver* discusses how to write a device driver that speaks the *vuid* protocol for a new input device.

- The client serializes multiple input devices being used by the user into a single stream of events.
- The client preserves the entire state of its input so that it may query this state.

Void Features

Void provides, among others, the following services to clients:

- A client may extend the capabilities of the predefined void by adding input devices. A client wants to be able to do this in a way that fits smoothly with its existing input paradigm.
- A client's code may be input device independent. A client can replace the physical device(s) underlying the virtual user input device and not have to change any input or event handlers, only the input device driver. In fact, the void interface doesn't care about physical devices. One physical device can masquerade as many logical devices and many physical devices can look like a single logical device.

Void Station Codes

This section defines the layout of the address space of void station codes. It explains how to extend the void address space for your own purposes. The meaning of void station codes is covered in the *Handling Input* chapter of the *SunView 1 Programmer's Guide*. The programmatic details of the void interface are covered in *Writing a Virtual User Input Device Driver* appendix to this document,

Address Space Layout

The address space for void events is 16 bits long, from 0 to 65535 inclusive. It is broken into 256 *segments* that are 256 entries long (VOID_SEG_SIZE). The top 8 bits contain a void segment identifier value. The bottom 8 bits contain a segment specific value from 0 to 255. Some segments have been predefined and some are available for expansion. Here is how the address space is currently broken down:

- ASCII_DEVID (0x00) — ASCII codes, which include META codes.
- TOP_DEVID (0x01) — Top codes, which are ASCII with the 9th bit on.
- Reserved (0x02 to 0x7B) — for Sun void implementations.
- SUNVIEW_DEVID (0x7C) — SunView semantic "action" events generated inside SunView by keymapping.
- PANEL_DEVID (0x7D) — Panel subwindow package event codes used internally in the panel package (see <suntool/panel.h>).
- SCROLL_DEVID (0x7E) — Scrollbar package event codes passed to scrollbar clients on interesting scrollbar activity (see <suntool/scrollbar.h>).

This device is a bit of a hodge-podge for historical reasons; the middle of the address space has SunView-related events in it (see `<sunwindow/win_input.h>`), and the virtual keyboard and virtual locator are thrown together.

Adding a New Segment

- `WORKSTATION_DEVID` (0x7F) — Virtual keyboard and locator (mouse) related event codes that describe a basic "workstation" device collection (see `<sundev/vuid_event.h>`).
- Reserved for Sun customers (0x80 to 0xFF) — if you are writing a new vuid, you can use a segment in here; see the next section.

`<sundev/vuid_event.h>` is the central registry of virtual user input devices. To allocate a new vuid you must modify this file:

- Choose an unused portion of the address space. Vuids from 0x00 to 0x7F are reserved for use by Sun. Vuids from 0x80 to 0xFF are reserved for Sun customers.
- Add the new device with a `*_DEVID` #define in this file. Briefly describe the purpose/usage of the device. Mention the place where more information can be found.
- Add the new device to the `Vuid_device` enumeration with a `VUID_devname` entry.
- List the specific event codes in **another** header file that is specific to the new device. `ASCII_DEVID`, `TOP_DEVID` and `WORKSTATION_DEVID` events are listed in `<sundev/vuid_event.h>` for historical reasons.

NOTE *A new vuid device can just as easily be a pure software construction as it can be a set of unique codes emitted by a new physical device driver.*

Input State Access

The complete state of the virtual input device is available. For example, one can ask questions about the up/down state of arbitrary keys.

```
int
win_get_vuid_value(windowfd, id)
    int    windowfd;
    short  id;
```

`id` is one of the event codes from `<sundev/vuid_event.h>` or `<sunwindow/win_input.h>`. `windowfd` can be any window file descriptor.

The result returned for keys is 0 for key is up and 1 for key is down; some vuid events return a range of numbers, such as mouse position. There is no error code for "no such key" because, by definition, the vuid event address space is the entire range of shorts and therefore you can't ask an incorrect question. 0 is the default event state.

Unencoded Input

Unencoded keyboard input is supported, for those customers who cannot use the normal keyboard input mechanism.

A new keyboard translation was introduced in 3.2. The type of translation is set by the `KIOCTRANS` ioctl (see `kb(4S)` and `kbd(4S)`). Old values were:

TR_NONE	for unencoded keyboard input outside the window system
TR_ASCII	for ASCII events (characters and escape sequences) outside the window system
TR_EVENT	for window input events inside the window system

A new value is now supported:

TR_UNTRANS_EVENT
gives unencoded keyboard values for input events inside the window system.

The client must have WIN_ASCII_EVENTS set in the window's input mask; if up-transitions are desired, WIN_UP_ASCII_EVENTS must also be set. (See Chapter 6, *Handling Input*, in the *SunView Programmer's Guide* for how to set input masks.)

The number of the key pressed or released will be passed in the event's `ie_code`, and the direction of the transition will be reported correctly by `event_is_up()` and `event_is_down()` (i.e., the NEGEVENT flag in `ie_flags` will be correct). The state of the shiftmask is undefined.

Events for other input (e.g. from the mouse) will be merged in the same stream with keyboard input, in standard window input fashion.

NOTE *Setting the keyboard translation has a global effect — it is not possible to get encoded input in one window and unencoded input in another on the same workstation.*

7.2. User Input Device Control

The number and kind of physical user input devices that can be used to drive SunView is open ended. Here are the controls for manipulating those devices.

Distinguished Devices

A keyboard and a mouse-like device are distinguished and settable directly. The *Desktops* chapter describes how they are specified in the screen structure. Here are two calls for changing them explicitly.

```
int
win_setkbd(windowfd, screen)
    int          windowfd;
    struct screen *screen;
```

changes the keyboard associated with `windowfd`'s desktop. Only the data pertinent to the keyboard is used (i.e., `screen->scr_kbdname`).

```
int
win_setms(windowfd, screen)
    int          windowfd;
    struct screen *screen;
```

changes the mouse associated with `windowfd`'s desktop. Only the data pertinent to the mouse is used (i.e., `screen->scr_msname`).

Arbitrary Devices

Arbitrary user input devices may be used to drive a workstation. However, some care must be exercised in selecting the combinations of devices. To install an input device with SunView, call `win_set_input_device()`.

```
int
win_set_input_device(windowfd, inputfd, name)
    int    windowfd;
    int    inputfd;
    char *name;
```

`windowfd` identifies (by association) the workstation on which the input device is to be installed. `name` is used to identify the device on subsequent calls to SunView, e.g., `/dev/kbd`. `name` may only be `SCR_NAMESIZE` characters long. Before calling this routine, open the input device and make any `ioctl(2)` calls to it to set it up to your requirements, e.g. possibly setting the speed of the serial port through which the device is coming in on. Pass the open file descriptor in as `inputfd`.

`win_set_input_device()` sends additional `ioctl(2)` calls to make the device operate as a Virtual User Input Device (if that is not its native mode) and operate in non-blocking read mode. The device's unread input is flushed. SunView starts reading from the device. Once `win_set_input_device()` returns, close `inputfd`. This action won't actually close the device; SunView has its own open file descriptor for the device.

Non-Vuid Devices

User input devices that only emit ASCII, and not vuid events, may be used by SunView. If the device does not respond to probing with the vuid `ioctls` SunView assumes it is an ASCII device and reads it one character at a time. Thus, SunView can handle input from a simple ASCII terminal without modification to any drivers. The routines in the section can be used with vuid or ASCII devices.

Device Removal

To remove an input device from SunView, call `win_remove_input_device()`.

```
int
win_remove_input_device(windowfd, name)
    int    windowfd;
    char *name;
```

`windowfd` identifies the workstation from which to remove the input device. `name` identifies the device. SunView resets the device to its original state.

Device Query

To ask if an input device is being utilized by a workstation, call `win_is_input_device()`.

```
int
win_is_input_device(windowfd, name)
    int    windowfd;
    char *name;
```

`windowfd` identifies the workstation being probed. `name` identifies the device. 0 is returned if the device is not being utilized, 1 is returned if it is, and -1 is

returned if there is an error.

Device Enumeration

To ask what all the input devices of the workstation are, call `win_enum_input_device()` which enumerates them all.

```
int
win_enum_input_device(windowfd, func, data)
    int      windowfd;
    int      (*func)();
    caddr_t   data;
```

`windowfd` identifies the workstation being probed. You pass the function `func` which is called once for every input device. The first argument passed to `func()` is a string which is the name of the device. The second argument passed to `func()` is `data`, which can be anything you want. If `func` returns something other than 0 the enumeration is terminated early.

`win_enum_input_device()` returns -1 if there was an error during the enumeration, 0 if it went smoothly and 1 if `func` terminated the enumeration early.

7.3. Focus Control

The concept of a split keyboard and pick input focus has been described in the *SunView 1 Programmer's Guide*. The user interface documentation describes it as "click to type" mode. It allows keyboard input events to be directed to a different window than the window that pick (cursor) inputs are sent to. Usually you want the keyboard input focus to stay in one window while the pick input focus is the window under the cursor.

Keyboard Focus Control

The following routine is called when a window gets a `KBD_REQUEST` event and the window doesn't need the keyboard focus.

```
win_refuse_kbd_focus(windowfd)
    int windowfd;
```

The following routine is used to change the keyboard focus. It is only a hint; the target window can refuse the keyboard focus or the user may not be running in click-to-type mode.

```
int
win_set_kbd_focus(windowfd, number)
    int windowfd, number;
```

`number` is the window that you want to have the keyboard focus.

The following routine gets the window number of the window that is currently the keyboard focus.

```
int
win_get_kbd_focus(windowfd)
    int windowfd;
```


Event Specification

This section describes how to programmatically specify which user actions are used as the focus control actions. The `sunview(1)` program has a set of flags to control the keyboard focus.

Setting the Caret Event

One of the ways to change the keyboard focus is to *set* the caret. Setting the focus passes the focus change event through to the application.

```
void
win_set_focus_event(windowfd, fe, shifts)
    int      windowfd;
    Firm_event *fe;
    int      shifts;
```

`windowfd` identifies the workstation. `fe` is a *firm event* pointer; the entire `Firm_event` structure is defined in the appendix titled *Writing a Virtual User Input Device Driver*. Only the `id` and the `value` fields are utilized in this call. The `id` field of `*fe` is set to the identifier of the event that is used to set the keyboard focus, e.g., `MS_LEFT`. The `value` field is set to the value of the event that is used to set the keyboard focus, e.g., 0 (up) or 1 (down). `shifts` is a mask of shift bits that indicate the required state of the shift keys needed in order to have the event described by `fe` treated as the keyboard focus change event. -1 means that you don't care. If you do care, use the same shift bits passed in the Event structure as discussed in the *Handling Input* chapter in the *SunView 1 Programmer's Guide*, e.g., `LEFTSHIFT`.

Getting the Caret Event

`win_get_focus_event()` returns the values set by `win_set_focus_event()`.

```
void
win_get_focus_event(windowfd, fe, shifts)
    int      windowfd;
    Firm_event *fe;
    int      *shifts;
```

`*fe` and `*shifts` are filled in with the current values.

Restoring the Caret

Another way to change the keyboard focus is to *restore* the caret. Restoring the focus swallows the focus change event so that it never makes it to the application. These two routines parallel the focus setting routines described above.

```
void
win_set_swallow_event(windowfd, fe, shifts)
    int      windowfd;
    Firm_event *fe;
    int      shifts;
```

```
void
win_get_swallow_event(windowfd, fe, shifts)
    int windowfd;
    Firm_event *fe;
    int *shifts;
```

7.4. Synchronization Control

This section discusses the concept of input synchronization in some detail. It's an important but subtle system mechanism. You should understand it fully before changing the system's default synchronization setting.

When running with input synchronization enabled, only one input event is being consumed, or processed, at a time. This event is called the *current event*. Ownership of the current event is bestowed by SunView to a single process; that process is said to have the *current event lock*. The lock belongs to a process, not a window device and is used to prevent any process from receiving an input event (via a `read(2)` system call) until the the lock has been released. This prevents race conditions between processes. This lets, for example, a user pop a window to the top and start typing to it before its image is drawn and have typing directed to the correct window. Input synchronization allows the process that currently has the lock to change its input mask and have the change recognized immediately so that applications won't miss events when they fall behind the user.

Input synchronization is not to be confused with the management of the input focus. The input focus is that window which is supposed to get the *next* input event and the input synchronization mechanism is used to determine when the *current* input event processing is completed.

The current event lock is acquired upon completing a read of a window device in which an input event was successfully read. For the duration of the lock, the current event lock owner decides what to do based on the current event and does it (or forks a process to do it). There is no notification to the input focus of input pending until the current event lock is released. Thus, there is typically only one process actively reading input at a time (except for rogue polling processes).

Releasing the Current Event Lock

When a process finishes with the current event, the current event lock is released via:

- A `read(2)` of the next event. If the next event is for the window that is doing the read then the lock is released and re-acquired immediately.
- A `select(2)` for input. This is the common case.
- An explicit `win_release_event_lock()` call (see below).

An explicit lock release call is appropriate for an application that knows that it no longer needs to query the state of the virtual input device or change its input mask and is about to do something moderately time consuming. Such an application can explicitly release the lock as soon as it recognizes that the event it has just read is not going to change event distribution.

```
win_release_event_lock(windowfd)
    int windowfd;
```

Current Event Lock Breaking

The current event lock is broken by SunView when the process with it visits the debugger or dies.

In addition, SunView explicitly breaks the current event lock if an application takes too long to process the event. The time is measured in process virtual time, not real time. This is a quiet lock breaking in that no message is displayed and

no signal is sent to the offending process. The duration of the time limit can be set, for the entire workstation, to a range of values:

- 0 — Windows have rampant race conditions. There is poor performance on high speed mouse tracking because the system can't compress mouse motion passed to applications.
- non-zero (approximately 1–10 seconds) — Most synchronization problems go away except when programs exceed the time limit. When SunView detects a process that exceeds the time limit the process temporarily goes into an unsynchronized mode until it catches up with the user.
- large-infinite (greater than 10 seconds) — Synchronization problems don't arise. Unfortunately, the user is locked into just one program at a time echoing/noticing input. The key combination **[Setup-I]** (the **[Setup]** key is the **[Stop]** key on Sun-2 and Sun-3 machines) explicitly breaks the lock.

Getting/Setting the Event Lock Timeout

The default time limit is 2 cpu seconds. You can get the current event lock timeout with a call to `win_get_event_timeout()`:

```
void
win_get_event_timeout(windowfd, tv)
    int                windowfd;
    struct timeval *tv;
```

*tv is filled in with the current value.

You can set the current event lock timeout via a call to `win_set_event_timeout()`:

```
void
win_set_event_timeout(windowfd, tv)
    int                windowfd;
    struct timeval *tv;
```

*tv is used as the current value.

7.5. Kernel Tuning Options

Some kernel tuning variable are settable using a debugger. However, you are advised not to change these unless you absolutely have to. You can look at the kernel with a debugger to see the default settings of these values, but you are playing with fire.

- `int ws_vq_node_bytes` is the number bytes to use for the input queue. You might increase this number if you find you are getting "Window input queue overflow!" and "Window input queue flushed!" messages. This needs to be modified before starting SunView in order to have any affect.
- `int ws_fast_timeout` is the number of hertz between polls of input devices when in fast mode. SunView polls its input devices at two speeds. The fast mode is the normal polling speed and the slow mode occurs when no action has been detected in the input devices for `ws_fast_poll_duration` hertz. This is all meant to save cpu cycles of useless polling when the user is not doing anything. The system is constantly bouncing between slow and fast polling mode.

`ws_fast_timeout` should never be 0. Decreasing this number improves interactive cursor tracking at the expense of increased system polling load.

- `int ws_slow_timeout` is the number of hertz between polls of input devices when in slow mode. `ws_slow_timeout` should never be 0. Decreasing this number improves interactive cursor tracking at the expense of increased system polling load.
- `int ws_fast_poll_duration` is discussed above. Increasing this number improves interactive performance at the expense of increased system polling load.
- `int ws_loc_still` is the number of hertz after which, if the locator has been still, a `LOC_STILL` event is generated.
- `struct timeval ws_lock_limit` is the process virtual time limit for a data or display lock. Increasing `ws_lock_limit` reduces the number of

... lock broken after time limit exceeded ...

console messages at the expense of slower response to dealing with lock hogs.

- `int ws_check_lock` and `struct timeval ws_check_time`
The check for `ws_lock_limit` doesn't start for `ws_check_lock` amount of real time after the lock is set. This is done to avoid system overhead for normal short lock intervals. Increasing `ws_check_lock` reduces system overhead on long lock holding situations at the expense of slower response to dealing with lock hogs.
- `int win_disable_shared_locking` is a flag that controls whether or not the window driver will try to reduce the overhead of display locking by using a shared memory mechanism. Even though there are no known problems with the shared memory locking mechanism, this variable is available as an escape hatch. If the window system leaves mouse cursor droppings, set this variable to 1. The default is 0. Setting this variable to 1 will result in reduced graphics performance.
- `int winlistcharsmax` is the maximum number of characters from the operating system's "character buffer" pool that SunView is willing to utilize. Upping this number can reduce "tossed" input situations. Turning on `wintossmmsg` (an `int`) will print a message if input has to be tossed. `winlistcharsmax` should only be increased by half again as much as its default.
- `int ws_set_favor` is a flag that controls whether or not the window driver will try to boost the priority of the window process (and its children) that has the current event lock. The default is 1. In very tight memory situations this dramatically improves interactive performance.

Changing the User Actions that Affect Input

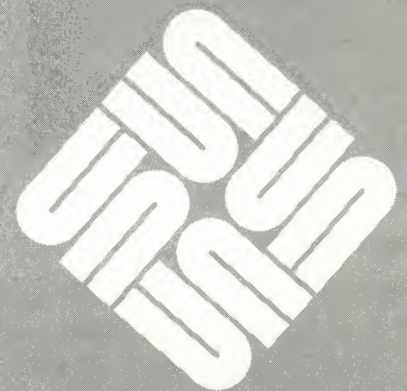
The following is provided so that you can change the user actions for the various real time interrupt actions.

```
typedef struct ws_usr_async {
    short    dont_touch1;
    short    first_id;      /* id of the 1st event */
    int      first_value;   /* value of the 1st event */
    short    second_id;     /* id of the 2nd event */
    int      second_value;  /* value of the 2nd event */
    int      dont_touch2;
} Ws_usr_async

Ws_usr_async ws_break_default = /* Event lock breaking */
    {0, SHIFT_TOP, 1, TOP_FIRST + 'i', 1, 0};
Ws_usr_async ws_stop_default = /* Stop event */
    {0, SHIFT_TOP, 1, SHIFT_TOP, 0, 0};
Ws_usr_async ws_flush_default = /* Input queue flushing */
    {0, SHIFT_TOP, 1, TOP_FIRST + 'f', 1, 0};
```


Advanced Notifier Usage

Advanced Notifier Usage	67
8.1. Overview	67
Contents	67
Viewpoint	67
Further Information	67
8.2. Notification	68
Client Events	68
Delivery Times	68
Handler Registration	68
The Event Handler	69
SunView Usage	69
Output Completed Events	69
Exception Occurred Events	70
Getting an Event Handler	70
8.3. Interposition	72
Registering an Interposer	72
Invoking the Next Function	73
Removing an Interposed Function	74
8.4. Posting	77
Client Events	77
Delivery Time Hint	77
Actual Delivery Time	77
Posting with an Argument	78



Storage Management	78
SunView Usage	79
Posting Destroy Events	80
Delivery Time	80
Immediate Delivery	80
Safe Delivery	80
8.5. Prioritization	81
The Default Prioritizer	81
Providing a Prioritizer	81
Dispatching Events	82
Getting the Prioritizer	83
8.6. Notifier Control	85
Starting	85
Stopping	85
Mass Destruction	85
Scheduling	86
Dispatching Clients	86
Getting the Scheduler	87
Client Removal	87
8.7. Error Codes	88
8.8. Restrictions on Asynchronous Calls into the Notifier	90
8.9. Issues	91

Advanced Notifier Usage

8.1. Overview

This chapter continues the description of the Notifier in *The Notifier* chapter of the *SunView 1 Programmer's Guide*.

Contents

This chapter presents areas which are not of general interest to the majority of SunView application programmers. These include:

- The registration of client, output and exception event handlers.
- Querying for the current address of one of a client's event handlers.
- Interposition of any event handler.
- Controlling the order in which a client receives events.
- Control over the dispatching of events.
- Controlling the order in which clients are notified.
- A list of error codes.
- Restrictions on calls into the Notifier.
- A list of open issues surrounding the Notifier.

Viewpoint

Although the Notifier falls under the umbrella of SunView, the Notifier can be looked at as a library package that is usable separately from the rest of SunView. The viewpoint of this chapter is one in which the Notifier stands alone from SunView. However, there are notes about SunView's usage of the Notifier throughout the chapter.

Further Information

You must read the chapter titled *The Notifier* in the *SunView 1 Programmer's Guide* before you tackle this chapter; it has information and examples about the Notifier and SunView's usage of it. In addition, *The Agent & Tiles* chapter in this manual has further information.

This split description of the Notifier may be a little awkward for advanced users of the Notifier but is much less confusing for the majority of users. You should refer to the index in the *SunView 1 Programmer's Guide* first and then the index in this book when using this material in a reference fashion.

8.2. Notification

This section presents the programming interface to the Notifier that clients use to register event handlers and receive notifications.

Links to the *SunView 1 Programmer's Guide*.

Only those areas not covered in the *Notifier* chapter of the *SunView 1 Programmer's Guide* are presented. In particular, *input pending* refers to the section in the other manual.

The two Notifier chapters are different in that the *SunView 1 Programmer's Guide* considers the Notifier in relation to SunView; thus, for example, in it notifier events are SunView Input Events, so their type is `Event *`. In this chapter, event is the more general type `Notify_event`.

Client Events

This section describes how *client events* are handled by the Notifier. From the Notifier's point of view, client events are defined and generated by the client. Client events are not interpreted by the Notifier in any way. The Notifier doesn't detect client events, it just detects UNIX-related events. The Notifier is responsible for dispatching client events to the client's event handler after the event has been *posted* with the Notifier by application code (see the section entitled *Posting* below).

Delivery Times

The Notifier normally sends client event notifications when it is *safe* to do so. This may involve some delay between when an event is posted and when it is delivered. However, a client may ask to always be *immediately* notified of the posting of a client event (see *Posting*, below).

Client Defined Signals

The immediate client event notification mechanism should be viewed as an extension of the UNIX signaling mechanism in which events are client defined signals. However, clients are strongly encouraged to only use safe client event handlers.

Handler Registration

To register a client event handler call:

```
Notify_func
notify_set_event_func(client, event_func, when)
    Notify_client      client;
    Notify_func        event_func;
    Notify_event_type  when;

enum notify_event_type {
    NOTIFY_SAFE=0,
    NOTIFY_IMMEDIATE=1,
};
typedef enum notify_event_type Notify_event_type;
```

when indicates whether the event handler will accept notifications only when it is safe (NOTIFY_SAFE) or at less restrictive times (NOTIFY_IMMEDIATE).¹⁶

¹⁶ For a rundown of the basics of registering event handlers see the section on *Event Handling* in the *Notifier* chapter of the *SunView 1 Programmer's Guide*.

The Event Handler

The calling sequence of a client event handler is:

```
Notify_value
event_func(client, event, arg, when)
    Notify_client    client;
    Notify_event     event;
    Notify_arg       arg;
    Notify_event_type when;
```

```
typedef caddr_t Notify_arg;
```

in which `client` is the client that called `notify_set_event_func()`. `event` is passed through from `notify_post_event()` (see *Posting*, below). `arg` is an additional argument whose type is dependent on the value of `event` and is completely defined by the client, like `event`. `when` is the actual situation in which `event` is being delivered (`NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`) and may be different from `when_hint` of `notify_post_event()`. The return value is one of `NOTIFY_DONE` or `NOTIFY_IGNORED`.

SunView Usage

You will almost certainly not need to directly register your own client event handler when using SunView. Window objects do this for themselves when they are created. However, note the following:

- A window has a client event handler that you may want to interpose in front of. See the section entitled *Monitoring and Modifying Window Behavior* in the *Notifier* chapter in the *SunView 1 Programmer's Guide*.
- SunView client event handlers are normally registered with `when` equal to `NOTIFY_SAFE`.
- The Agent reads input events from a window's file descriptor and posts them to the client via the `win_post_event()` call. See the section titled *Notifications From the Agent* in *The Agent & Tiles* chapter.

Output Completed Events

Notifications for output completed notifications are similar to input pending notifications, covered in the chapter on the Notifier in the *SunView 1 Programmer's Guide*.

```
Notify_func
notify_set_output_func(client, output_func, fd)
    Notify_client    client;
    Notify_func      output_func;
    int              fd;
```

```
Notify_value
output_func(client, fd)
    Notify_client    client;
    int              fd;
```

Exception Occurred Events

Exception occurred notifications are similar to input pending notifications. The only known devices that generate exceptions at this time are stream-based socket connections when an out-of-band byte is available. Thus, a SIGURG signal catcher is set up by the Notifier, much like SIGIO for asynchronous input.

```
Notify_func
notify_set_exception_func(client, exception_func, fd)
    Notify_client  client;
    Notify_func    exception_func;
    int            fd;
```

```
Notify_value
exception_func(client, fd)
    Notify_client  client;
    int            fd;
```

Getting an Event Handler

Here is the list of routines that allow you to retrieve the value of a client's event handler. The arguments to each `notify_get_*_func()` function parallel the associated `notify_set_*_func()` function described elsewhere except for the absence of the event handler function pointer. Thus, we don't describe the arguments in detail here. Refer back to the associated `notify_set_*_func()` descriptions for details.¹⁷

A return value of `NOTIFY_FUNC_NULL` indicates an error. If `client` is unknown then `notify_errno` is set to `NOTIFY_UNKNOWN_CLIENT`. If no event handler is registered for the specified event then `notify_errno` is set to `NOTIFY_NO_CONDITION`. Other values of `notify_errno` are possible, depending on the event, e.g., `NOTIFY_BAD_FD` if an invalid file descriptor is specified (see the associated `notify_set_*_func()`).

Here is a list of event handler retrieval routines:

```
Notify_func
notify_get_input_func(client, fd)
    Notify_client  client;
    int            fd;
```

```
Notify_func
notify_get_event_func(client, when)
    Notify_client  client;
    Notify_event_type  when;
```

```
Notify_func
notify_get_output_func(client, fd)
    Notify_client  client;
    int            fd;
```

```
Notify_func
```

¹⁷ It is recommended that you use the Notifier's interposition mechanism instead of trying to do interposition yourself using these `notify_get_*_func()` routines.


```
notify_get_exception_func(client, fd)
    Notify_client  client;
    int            fd;

Notify_func
notify_get_itimer_func(client, which)
    Notify_client  client;
    int            which;

Notify_func
notify_get_signal_func(client, signal, mode)
    Notify_client  client;
    int            signal;
    Notify_signal_mode  mode;

Notify_func
notify_get_wait3_func(client, pid)
    Notify_client  client;
    int            pid;

Notify_func
notify_get_destroy_func(client)
    Notify_client  client;
```

8.3. Interposition

There are many reasons why an application might want to interpose a function in the call path to a client's event handler:

- An application may want to use the fact that a client has received a particular notification as a trigger for some application-specific processing.
- An application may want to filter the notifications to a client, thus modifying the client's behavior.
- An application may want to extend the functionality of a client by handling notifications that the client is not programmed to handle.

The Notifier supports interposition by keeping track of how interposition functions are ordered for each type of event for each client. Here is a typical example of interposition:

- An application creates a client. The client has set up its own client event handler using `notify_set_event_func()`.
- The application tells the Notifier that it wants to interpose its function in front of the client's event handler by calling `notify_interpose_event_func()` (described below).
- When the application's interposed function is called, it tells the Notifier to call the next function, i.e., the client's function, via a call to `notify_next_event_func()` (described below).

Registering an Interposer

The following routines let you interpose your own function in front of a client's event handler. The arguments to each `notify_interpose_*_func()` function parallel the associated `notify_set_*_func()` function described above. Thus, we don't describe the arguments in detail here. Refer back to the associated `notify_set_*_func()` descriptions for details.

NOTE *The one exception to this rule is that the arguments to `notify_interpose_itimer_func()` are a subset of the arguments to `notify_set_itimer_func()`.*

```

Notify_error
notify_interpose_input_func(client, input_func, fd)
    Notify_client  client;
    Notify_func    input_func;
    int fd;

Notify_error
notify_interpose_event_func(client, event_func, when)
    Notify_client  client;
    Notify_func    event_func;
    Notify_event_type  when;

Notify_error
notify_interpose_output_func(client, output_func, fd)
    Notify_client  client;
    Notify_func    output_func;
    int            fd;

```



```

Notify_error
notify_interpose_exception_func(client, exception_func, fd)
    Notify_client  client;
    Notify_func    exception_func;
    int            fd;

```

```

Notify_error
notify_interpose_itimer_func(client, itimer_func, which)
    Notify_client  client;
    Notify_func    itimer_func;
    int            which;

```

```

Notify_error
notify_interpose_signal_func(client, signal_func,
                             signal, mode)
    Notify_client  client;
    Notify_func    signal_func;
    int            signal;
    Notify_signal_mode  mode;

```

```

Notify_error
notify_interpose_wait3_func(client, wait3_func, pid)
    Notify_client  client;
    Notify_func    wait3_func;
    int            pid;

```

```

Notify_error
notify_interpose_destroy_func(client, destroy_func)
    Notify_client  client;
    Notify_func    destroy_func;

```

The return values from these functions may be one of:

- NOTIFY_OK — The interposition was successful.
- NOTIFY_UNKNOWN_CLIENT — `client` is not known to the Notifier.
- NOTIFY_NO_CONDITION — There is no event handler of the type specified.
- NOTIFY_FUNC_LIMIT — The current implementation allows five levels of interposition for every type of event handler, the original event handler registered by the client plus five interposers. NOTIFY_FUNC_LIMIT indicates that this limit has been exceeded.

If the return value is something other than NOTIFY_OK then `notify_errno` contains the error code.

Invoking the Next Function

Here is the list of routines that you call from your interposed function in order to invoke the next function in the interposition sequence. The arguments and return value of each `notify_next_*_func()` function are the same as the arguments passed to the your interposer function. Thus, we don't describe the arguments in detail here. Refer back to the associated event handler descriptions for details.

```
Notify_value
notify_next_input_func(client, fd)
    Notify_client  client;
    int           fd;

Notify_value
notify_next_event_func(client, event, arg, when)
    Notify_client  client;
    Notify_event   *event;
    Notify_arg     arg;
    Notify_event_type when;

Notify_value
notify_next_output_func(client, fd)
    Notify_client  client;
    int           fd;

Notify_value
notify_next_exception_func(client, fd)
    Notify_client  client;
    int           fd;

Notify_value
notify_next_itimer_func(client, which)
    Notify_client  client;
    int           which;

Notify_value
notify_next_signal_func(client, signal, mode)
    Notify_client  client;
    int           signal;
    Notify_signal_mode mode;

Notify_value
notify_next_wait3_func(client, pid, status, rusage)
    Notify_client  client;
    union          wait status;
    struct rusage  rusage;
    int           pid;

Notify_value
notify_next_destroy_func(client, status)
    Notify_client  client;
    Destroy_status status;
```

Removing an Interposed Function

Here is the list of routines that allow you to remove the interposer function that you installed using a `notify_interpose_*_func()` call. The arguments to each `notify_remove_*_func()` function is exactly the same as the associated `notify_set_*_func()` function described above. Thus, we don't describe the arguments in detail here.

NOTE *The one exception to this rule is that the arguments to `notify_remove_itimer_func()` are a subset of the arguments to `notify_set_itimer_func()`.*

```
Notify_error
notify_remove_input_func(client, input_func, fd)
    Notify_client  client;
    Notify_func    input_func;
    int            fd;
```

```
Notify_error
notify_remove_event_func(client, event_func, when)
    Notify_client  client;
    Notify_func    event_func;
    Notify_event_type  when;
```

```
Notify_error
notify_remove_output_func(client, output_func, fd)
    Notify_client  client;
    Notify_func    output_func;
    int            fd;
```

```
Notify_error
notify_remove_exception_func(client, exception_func, fd)
    Notify_client  client;
    Notify_func    exception_func;
    int            fd;
```

```
Notify_error
notify_remove_itimer_func(client, itimer_func, which)
    Notify_client  client;
    Notify_func    itimer_func;
    int            which;
```

```
Notify_error
notify_remove_signal_func(client, signal_func, signal, mode)
    Notify_client  client;
    Notify_func    signal_func;
    int            signal;
    Notify_signal_mode  mode;
```

```
Notify_error
notify_remove_wait3_func(client, wait3_func, pid)
    Notify_client  client;
    Notify_func    wait3_func;
    int            pid;
```

```
Notify_error
notify_remove_destroy_func(client, destroy_func)
    Notify_client  client;
    Notify_func    destroy_func;
```


If the return value is something other than `NOTIFY_OK` then `notify_errno` contains the error code. The error codes are the same as those associated with `notify_interpose_*_func()` calls.

8.4. Posting

This section describes how to post client events and destroy events with the Notifier.

Client Events

A client event may be posted with the Notifier at any time. The poster of a client event may suggest to the Notifier when to deliver the event, but this is only a hint. The Notifier will see to it that it is delivered at an appropriate time (more on this below). The call to post a client event is:

```
typedef char * Notify_event;

Notify_error
notify_post_event(client, event, when_hint)
    Notify_client      client;
    Notify_event       event;
    Notify_event_type  when_hint;
```

The client handle from `notify_set_event_func()` is passed to `notify_post_event()`. `event` is defined and interpreted solely by the client. A return code of `NOTIFY_OK` indicates that the notification has been posted. Other values indicate an error condition. `NOTIFY_UNKNOWN_CLIENT` indicates that `client` is unknown to the Notifier. `NOTIFY_NO_CONDITION` indicates that `client` has no client event handler registered with the Notifier.

Delivery Time Hint

Usually it is during the call to `notify_post_event()` that the client event handler is called. Sometimes, however, the notification is queued up for later delivery. The Notifier chooses between these two possibilities by noting which kinds of client event handlers `client` has registered, whether it is safe and what the value of `when_hint` is. Here are the cases broken down by which kinds of client event handlers `client` has registered:

- **Immediate only** — Whether `when_hint` is `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE` the event is delivered immediately.
- **Safe only** — Whether `when_hint` is `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE` the event is delivered when it is safe.
- **Both safe and immediate** — A client may have both an immediate client event handler as well as a safe client event handler. If `when_hint` is `NOTIFY_SAFE` then the notification is delivered to the safe client event handler when it is safe. If `when_hint` is `NOTIFY_IMMEDIATE` then the notification is delivered to the immediate client event handler right away. If the immediate client event handler returns `NOTIFY_IGNORED` then the same notification will be delivered to the safe client event handler when it is safe.

Actual Delivery Time

For client events, other than knowing which event handler to call, the main function of the Notifier is to know when to make the call. The Notifier defines when it is safe to make a client notification. If it is not safe, then the event is queued up for later delivery. Here are the conventions:

- A client that has registered an immediate client event handler is sent a notification as soon as it is received. The client has complete responsibility

for handling the event safely. It is rarely safe to do much of anything when an event is received asynchronously. Usually, just setting a flag that indicates that the event has been received is about the safest thing that can be done.

- A client that has registered a safe client event handler will have a notification queued up for later delivery when the notification was posted during an asynchronous signal notification. Immediate delivery is not safe because your process, just before receiving the signal, may have been executing code at any arbitrary place.
- A client that has registered a safe client event handler will have a notification queued up for later delivery if the client's safe client event handler hasn't returned from processing a previous event. This convention is mainly to prevent the cycle: Notifier notifies *A*, who notifies *B*, who notifies *A*. *A* could have had its data structures torn up when it notified *B* and was not in a state to be reentered.

Implied in these conventions is that a safe client event handler is called immediately from other UNIX event handlers. For example:

- A client's input pending event handler is called by the Notifier.
- Two characters are read by the client's input pending event handler.
- The first character is given to the Notifier to deliver to the client's safe event handler.
- The Notifier immediately delivers the character to the client event handler.
- Returning back to the input pending event handler, the second character is sent. This character is also delivered immediately.

Posting with an Argument

SunView posts a fixed field structure with each event. Sometimes additional data must be passed with an event. For instance when the scrollbar posts an event to its owner to do a scroll. The scrollbar's handle is passed as an argument along with the event. `notify_post_event_and_arg()` provides this argument passing mechanism (see below).

Storage Management

When posting a client event there is the possibility of delivery being delayed. In the case of SunView, the event being posted is a pointer to a structure. The Notifier avoids an invalid (dangling) pointer reference by copying the event if delivery is delayed. It calls routines the client supplies to copy the event information and later to free up the storage the copy uses.

`notify_post_event_and_arg()` provides this storage management mechanism.


```

Notify_error
notify_post_event_and_arg(client, event, when_hint, arg,
                           copy_func, release_func)
    Notify_client    client;
    Notify_event     event;
    Notify_event_type when_hint;
    Notify_arg       arg;
    Notify_copy      copy_func;
    Notify_release   release_func;

typedef caddr_t Notify_arg;

typedef Notify_arg (*Notify_copy)();
#define NOTIFY_COPY_NULL ((Notify_copy)0)

typedef void (*Notify_release)();
#define NOTIFY_RELEASE_NULL ((Notify_release)0)

```

`copy_func()` is called to copy `arg` (and optionally `event`) when `event` and `arg` needed to be queued for later delivery. `release_func()` is called to release the storage allocated during the copy call when `event` and `arg` were no longer needed by the Notifier.

Any of `arg`, `copy_func()` or `release_func()` may be null. If `copy_func` is not `NOTIFY_COPY_NULL` and `arg` is `NULL` then `copy_func()` is called anyway. This allows event the opportunity to be copied because `copy_func()` takes a pointer to `event`. The pointed to `event` may be replaced as a side affect of the copy call. The same applies to a non-`NOTIFY_RELEASE_NULL` release function with a `NULL` `arg` argument.

The `copy()` and `release()` routines are client-dependent so you must write them yourself. Their calling sequences follow:

```

Notify_arg
copy_func(client, arg, event_ptr)
    Notify_client    client;
    Notify_arg       arg;
    Notify_event     *event_ptr;

void
release_func(client, arg, event)
    Notify_client    client;
    Notify_arg       arg;
    Notify_event     event;

```

SunView Usage

There are Agent calls to post an event to a tile that provide a layer over the posting calls described here (see `win_post_event()` in the chapter entitled *The Agent & Tiles*).

Posting Destroy Events

When a destroy notification is set, the Notifier also sets up a synchronous signal condition for SIGTERM that will generate a DESTROY_PROCESS_DEATH destroy notification. Otherwise, a destroy function will not be called automatically by the Notifier. One or two (depending on whether the client can veto your notification) explicit calls to `notify_post_destroy()` need be made.

```
Notify_error
notify_post_destroy(client, status, when)
    Notify_client      client;
    Destroy_status     status;
    Notify_event_type  when;
```

NOTIFY_INVALID is returned if status or when is not defined. After notifying a client to destroy itself, all references to client are purged from the Notifier.

Delivery Time

Unlike a client event notification, the Notifier doesn't try to detect when it is safe to post a destroy notification. Thus, a destroy notification can come at any time. It is up to the good judgement of a caller of `notify_post_destroy()` or `notify_die()` (described in the section titled *Notifier Control*) to make the call at a time that a client is not likely to be in the middle of accessing its data structures.

Immediate Delivery

If status is DESTROY_CHECKING and when is NOTIFY_IMMEDIATE then `notify_post_destroy()` may return NOTIFY_DESTROY_VETOED if the client doesn't want to go away.

Safe Delivery

Often you want to tell a client to go away at a safe time. This implies that delivery of the destroy event will be delayed, in which case the return value of `notify_post_destroy()` can't be NOTIFY_DESTROY_VETOED because the client hasn't been asked yet. To get around this problem the Notifier will flush the destroy event of a checking/destroy pair of events if the checking phase is vetoed. Thus, a common idiom is:

```
(void) notify_post_destroy(client, DESTROY_CHECKING,
                           NOTIFY_SAFE);
(void) notify_post_destroy(client, DESTROY_CLEANUP,
                           NOTIFY_SAFE);
```


8.5. Prioritization

The order in which a particular client's conditions are notified may be controlled by providing a *prioritizer* operation.¹⁸

The Default Prioritizer

The default prioritizer makes its notifications in this order (any asynchronous or immediate notifications have already been sent):

- Interval timer notifications (ITIMER_REAL and then ITIMER_VIRTUAL).
- Child process control notifications.
- Synchronous signal notifications by ascending signal numbers.
- Exception file descriptor activity notifications by ascending fd numbers.
- Handle client events by order in which received.
- Output file descriptor activity notifications by ascending fd numbers.
- Input file descriptor activity notifications by ascending fd numbers.

Providing a Prioritizer

This section describes how a client can provide its own prioritizer.

```
Notify_func
notify_set_prioritizer_func(client, prioritizer_func)
    Notify_client  client;
    Notify_func    prioritizer_func;
```

`notify_set_prioritizer_func()` takes an opaque client handle and the function to call before any notifications are sent to `client`. The previous function that would have been called is returned. If this function was never defined then the default prioritization function is returned. If the `prioritizer_func()` argument is `NOTIFY_FUNC_NULL` then no client prioritization is done for `client` and the default prioritizer is used.

The calling sequence of a prioritizer function is:

```
Notify_value
prioritizer_func(client, nfd, ibits_ptr, obits_ptr,
                 ebits_ptr, nsig, sigbits_ptr, auto_sigbits_ptr,
                 event_count_ptr, events, args)
    Notify_client  client;
    fd_set        *ibits_ptr, *obits_ptr, *ebits_ptr;
    int           nfd, nsig, *sigbits_ptr,
                 *auto_sigbits_ptr, *event_count_ptr;
    Notify_event   *events;
    Notify_arg      *args;
#define SIG_BIT(sig)  (1 << ((sig)-1))
```

in which `client` from `notify_set_prioritizer_func()` are passed to `prioritizer_func()`. In addition, all the notifications that the Notifier is planning on sending to `client` are described in the other parameters. This data reflects only data that `client` has expressed interest in by asking for

¹⁸ It is anticipated that this facility will be rarely used by clients and that a client will rely on the ordering provided by the default prioritizer.

notification of these conditions.

`nfd` describes the maximum number of valid bits in the `fd_set` structures¹⁹ pointed to by `ibits_ptr`, `obits_ptr`, and `ebits_ptr`. `ibits_ptr` points to a bit mask of those file descriptors with input pending for `client`; similarly `obits_ptr` points to a bit mask of file descriptors with output completed, and `ebits_ptr` points to a bit mask of file descriptors on which an exception occurred. `nsig` describes the maximum number of valid bits in the arrays pointed to by `sigbits_ptr` and `auto_sigbits_ptr`. `sigbits_ptr` is a bit mask of signals received for which `client` has a condition registered; the `SIG_BIT` macro can be used to access the correct bit. `auto_sigbits_ptr` is a bit mask of signals received that the Notifier is managing on behalf of `client`. `event_count` is the number of events in the array `events`. `events` is an array of pending client events and `args` is the parallel array of event arguments.

The return value is one of `NOTIFY_DONE` or `NOTIFY_IGNORED`. These have their normal meanings:

- `NOTIFY_DONE` — All of the conditions had notifications sent for them. This implies that no further notifications should be sent to `client` this time around the notification loop. Unsent notifications are preserved for consideration the next time around the notification loop.
- `NOTIFY_IGNORED` — A notification was not sent for one or more of the conditions, i.e., some notifications may have been sent, but not all. This implies that another prioritizer should try to send any remaining notifications to `client`.

Dispatching Events

From within a prioritization routine, the following functions are called to cause the specified notifications to be sent:

```
Notify_error
notify_event(client, event, arg)
    Notify_client  client;
    Notify_event   event;
    Notify_arg     arg;
```

```
Notify_error
notify_input(client, fd)
    Notify_client  client;
    int           fd;
```

```
Notify_error
notify_output(client, fd)
    Notify_client  client;
    int           fd;
```

```
Notify_error
```

¹⁹ With the increase past 32 of the maximum number of file descriptors under SunOS Release 4.0, the masks of FD bits are no longer `ints` but a special structure, defined in `<sys/types.h>`.

```

notify_exception(client, fd)
    Notify_client  client;
    int           fd;

Notify_error
notify_itimer(client, which)
    Notify_client  client;
    int           which;

Notify_error
notify_signal(client, signal)
    Notify_client  client;
    int           signal;

Notify_error
notify_wait3(client)
    Notify_client  client;

```

The Notifier won't send any notifications that it wasn't planning on sending anyway, so one can't use these calls to drive clients programmatically. A return value of `NOTIFY_OK` indicates that `client` was sent the notification. A return value of `NOTIFY_UNKNOWN_CLIENT` indicates that `client` is not recognized by the Notifier and no notification was sent. A return value of `NOTIFY_NO_CONDITION` indicates that `client` does not have the requested notification pending and no notification was sent.

A client may choose to replace the default prioritizer. Alternatively, a client's prioritizer may call the default prioritizer after sending only a few notifications. Any notifications not explicitly sent by a client prioritizer will be sent by the default prioritizer (when called), in their normal turn. Once notified, a client will not receive a duplicate notification for the same event.

Signals indicated by bits in `sigbits_ptr` should call `notify_signal()`. Signals in `auto_sigbits_ptr` need special treatment:

- `SIGALRM` means that `notify_itimer()` should be called with a `which` of `ITIMER_REAL`.
- `SIGVTALRM` means that `notify_itimer()` should be called with a `which` of `ITIMER_VIRTUAL`.
- `SIGCHLD` means that `notify_wait3()` should be called.

Asynchronous signal notifications, destroy notifications and client event notifications that were delivered right when they were posted do not pass through the prioritizer.

Getting the Prioritizer

`notify_get_prioritizer_func()` returns the current prioritizer of a client.

```

Notify_func
notify_get_prioritizer_func(client)
    Notify_client  client;

```

`notify_get_prioritizer_func()` takes an opaque client handle. The

function that will be called before any notifications are sent to `client` is returned. If this function was never defined for `client` then a default function is returned. A return value of `NOTIFY_FUNC_NULL` indicates an error. If `client` is unknown then `notify_errno` is set to `NOTIFY_UNKNOWN_CLIENT`.

8.6. Notifier Control

The following are the Notifier wide (vs single condition) operations.

Starting

Here is the routine for starting the notification loop of the Notifier:

```
Notify_error
notify_start()
```

This is the main control loop. It is usually called from the main routine of your program after all the clients in your program have registered their event handlers with the Notifier.²⁰ The return values are:

- NOTIFY_OK — Terminated normally by `notify_stop()` (see below).
- NOTIFY_NO_CONDITION — There are no conditions registered with the Notifier.
- NOTIFY_INVALID — Tried to call `notify_start()` before returned from original call, i.e., this call is not reentrant.
- NOTIFY_BADF — One of the file descriptors in one of the conditions is not valid.

Stopping

An application may want to break the Notifier out its main loop after the Notifier finishes sending any pending notifications.

```
Notify_error
notify_stop()
```

This causes `notify_start()` to return. The return values are NOTIFY_OK (will terminate `notify_start()`) and NOTIFY_NOT_STARTED (`notify_start()` not entered).

Mass Destruction

The following routine causes the all client destruction functions to be called immediately with status:

```
Notify_error
notify_die(status)
    Destroy_status status;
```

This causes the all client destruction functions to be called immediately with status as the reason. The return values are NOTIFY_OK or NOTIFY_DESTROY_VETOED; the latter indicates that someone called `notify_veto_destroy()` and status was DESTROY_CHECKING. It is then the responsibility of the caller of `notify_die()` to exit the process, if so desired. See the discussion on `notify_post_destroy()` for more information.

²⁰ SunView programs usually call `window_main_loop()` instead of `notify_start()`.

Scheduling

There is the mechanism for controlling the order in which clients are notified. (Controlling the order in which a particular client's notifications are sent to it is done by that client's prioritizer operation; see the *Prioritization* section earlier.)

```
Notify_func
notify_set_scheduler_func(scheduler_func)
    Notify_func scheduler_func;
```

`notify_set_scheduler_func()` allows you to arrange the order in which clients are called. (Individual clients can control the order in which their event handlers are called by setting up prioritizers.)

`notify_set_scheduler_func()` takes a function to call to do the scheduling of clients. The previous function that would have been called is returned. This returned function will (almost always) be important to store and call later because it is most likely the default scheduler.

Replacement of the default scheduler will be done most often by a client that needs to make sure that other clients don't take too much time servicing all of their notifications. For example, if doing "real-time" cursor tracking in a user process, the tracking client wants to schedule itself ahead of other clients whenever there is input pending on the mouse.

The calling sequence of a scheduler function is:

```
Notify_value
scheduler_func(n, clients)
    int n;
    Notify_client *clients;
```

in which a list of `n` clients, all of which are slated to receive some notification this time around, are passed into `scheduler_func()`. The scheduler scans `clients` and makes calls to `notify_client()` (see below). Clients so notified should have their slots in `clients` set to `NOTIFY_CLIENT_NULL`. The return value from `scheduler_func()` is one of:

- `NOTIFY_DONE` — All of the clients had a chance to send notifications. This implies that no further clients should be scheduled this time around the notification loop. Unsent notifications are preserved for consideration the next time around the notification loop.
- `NOTIFY_IGNORED` — One or more `clients` were scheduled, i.e., some `clients` may have been scheduled, but not all. This implies that another scheduler should try to schedule any clients in `clients` that are not `NOTIFY_CLIENT_NULL`.

Dispatching Clients

The following routine is called from scheduler routines to cause all the pending notifications for `client` to be sent:

```
Notify_error
notify_client(client)
    Notify_client client;
```

The return value is one of `NOTIFY_OK` (client notified) or `NOTIFY_NO_CONDITION` (no conditions for client, perhaps `notify_client()` was already called with this client handle) or

NOTIFY_UNKNOWN_CLIENT (unknown client).

Getting the Scheduler

The following routine returns the function that will be called to do client scheduling:

```
Notify_func  
notify_get_scheduler_func()
```

This function is always defined to at least be the default scheduler.

Client Removal

A client can remove itself from the control of the Notifier with

`notify_remove()`:

```
Notify_error  
notify_remove(client)  
    Notify_client client;
```

`notify_remove()` is a utility to allow easy removal of a client from the Notifier's control. All references to `client` are purged from the Notifier. This routine is almost always called by the client itself. The return values are NOTIFY_OK (success) and NOTIFY_UNKNOWN_CLIENT (unknown client).

8.7. Error Codes

This section describes the basic error handling scheme used by the Notifier and lists the meaning of each of the possible error codes. Every call to the Notifier returns a value that indicates success or failure. On an error condition, `notify_errno` describes the failure. `notify_errno` is set by the Notifier much like `errno` is set by UNIX system calls, i.e., `notify_errno` is set only when an error is detected during a call to the Notifier and is not reset to `NOTIFY_OK` on a successful call to the Notifier.

```
enum notify_error {
    ... /* Listed below */
};
typedef enum notify_error Notify_error;

extern Notify_error notify_errno;
```

Here is a complete list of error codes:

- `NOTIFY_OK` — The call was completed successfully.
- `NOTIFY_UNKNOWN_CLIENT` — The client argument is not known by the Notifier. A `notify_set_*_func` type call need be done in order for the Notifier to recognize a client.
- `NOTIFY_NO_CONDITION` — A call was made to access the state of a condition but the condition is not set with the Notifier for the given client. This can arise when a `notify_get_*_func()` type call was done before the equivalent `notify_set_*_func()` call was done. Also, the Notifier automatically clears some conditions after they have occurred, e.g., when an interval timer expires.
- `NOTIFY_BAD_ITIMER` — The `which` argument to an interval timer routine was not valid.
- `NOTIFY_BAD_SIGNAL` — The `signal` argument to a signal routine was out of range.
- `NOTIFY_NOT_STARTED` — A call to `notify_stop()` was made but the Notifier was never started.
- `NOTIFY_DESTROY_VETOED` — A client refused to be destroyed during a call to `notify_die()` or `notify_post_destroy()` when status was `DESTROY_CHECKING`.
- `NOTIFY_INTERNAL_ERROR` — This error code indicates some internal inconsistency in the Notifier itself has been detected.
- `NOTIFY_SRCH` — The `pid` argument to a child process control routine was not valid.
- `NOTIFY_BADF` — The `fd` argument to an input or output routine was not valid.
- `NOTIFY_NOMEM` — The Notifier dynamically allocates memory from the heap. This error code is generated if the allocator could not get any more memory.

- NOTIFY_INVALID — Some argument to a call to the Notifier contained an invalid argument.
- NOTIFY_FUNC_LIMIT — An attempt to set an interposer function has encountered the limit of the number of interposers allowed for a single condition.

The routine `notify_perror()` acts just as the library call `perror(3)`.

```
notify_perror(str)
    char *str;
```

`notify_perror()` prints the string `str`, followed by a colon and followed by a string that describes `notify_errno` to `stderr`.

8.8. Restrictions on Asynchronous Calls into the Notifier

The Notifier takes precautions to protect its data against corruption during calls into it while it is calling out to an asynchronous/immediate event handler. The Notifier may issue an asynchronous notification for an asynchronous signal condition, an immediate client event condition or a destroy condition. Most calls from event handlers back into the Notifier are permitted, but there are some restrictions:

- Some calls are not permitted. In particular, they are:

```
notify_start()  
notify_client()
```

- Only a certain number of calls into the Notifier are permitted. This restriction is due to how the Notifier handles memory management in a safe way during asynchronous processing. As a guideline, do not do more than five calls of the `notify_set_*_func()`, `notify_interpose_*_func()` or `notify_post_*()` variety during an asynchronous notification.
- The Notifier is not prepared to handle calls into it from signal catching routines that a client has set up with `signal(3)` or `sigvec(2)`.

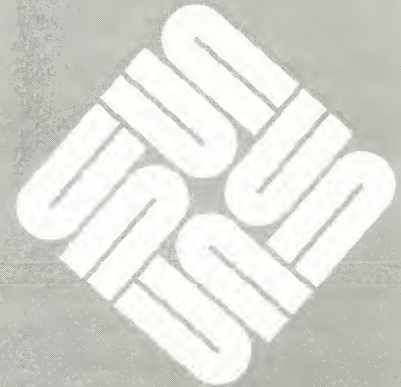
8.9. Issues

Here are some issues surrounding the Notifier:

- The layer over the UNIX signal mechanism is not complete. Signal blocking (`sigblock(2)`) can still safely be done in the flow of control of a client to protect critical portions of code as long as the previous signal mask is restored before returning to the Notifier. Signal pausing (`sigpause(2)`) is essentially done by the Notifier. Signal masking (`sigmask(2)`) can be accomplished via multiple `notify_set_signal_func()` calls. Setting up a process signal stack (`sigstack(2)`) can still be done. Setting the signal catcher mask and on-signal-stack flag (`sigvec(2)`) could be done by reaching around the Notifier but is not supported.
- Not all process resources are multiplexed (e.g., `rlimit(2)`, `setjmp(2)`, `umask(2)`, `setquota(2)`, and `setpriority(2)`), only ones that have to do with flow of control multiplexing. Thus, some level of cooperation and understanding need exist between packages in the single process.
- One can make a case for intercepting `close(2)` and `dup(2)` calls so that the Notifier is not waiting on invalid or incorrect file descriptors if a client forgets to remove its conditions from the Notifier before making these calls.
- One can make a case for intercepting `signal(3)` and `sigvec(2)` calls so that the Notifier doesn't get confused by programs that fail to use the Notifier to manage its signals.
- One can make a case for intercepting `setitimer(2)` calls so that the Notifier doesn't get confused by programs that fail to use the Notifier to manage interval timers.
- One can make a case for intercepting `ioctl(2)` calls so that the Notifier doesn't get fouled up by programs that use `FIONBIO` and `FIOASYNC` instead of the equivalent `fcntl(2)` calls.
- One can make a case for intercepting `readv(2)` and `write(2)` just like `read(2)` and `select(2)` so that a program doesn't tie up the process.
- The Notifier is not a lightweight process mechanism that maintains a stack per thread of control. However, if such a mechanism becomes available then the Notifier will still be valuable for its support of notification-based clients.
- Client events are disjoint from UNIX events. This is done to give complete freedom to clients as to how events are defined. One could imagine certain clients wanting to unify client and UNIX events. This could be done with a layer of software on top of the Notifier. A client could define events as pointers to structures that contain event codes and event specific arguments. The event codes would include the equivalents of UNIX event notifications. The event specific arguments would contain, for example, the file descriptor of an input-pending notification. When an input-pending notification from the the Notifier was sent to a client, the client would turn around and post the equivalent client event notification.
- One could imagine extending the Notifier to provide a record and replay mechanism that would drive an application. However, this is not supported by the current interface.

The Selection Service and Library

The Selection Service and Library	95
9.1. Introduction	95
9.2. Basic Concepts	96
9.3. Fast Overview	96
The Selection Service vs. the Selection Library	97
9.4. Selection Service Protocol: an Example	97
Secondary Selection Between Two Windows	97
Notes on the Above Example	101
9.5. Topics in Selection Processing	101
Reporting Function-Key Transitions	101
Sending Requests to Selection Holders	102
Long Request Replies	104
Acquiring and Releasing Selections	104
Callback Procedures: Function-Key Notifications	104
Responding to Selection Requests	105
Callback Procedures: Replying to Requests	106
9.6. Debugging and Administrative Facilities	109
9.7. Other Suggestions	109
If You Die	109
9.8. Reference Section	110
Required Header Files	110
Enumerated Types	110
Other Data Definitions	110



Procedure Declarations	112
9.9. Common Request Attributes	121
9.10. Two Program Examples	125
<i>get_selection</i> Code	125
<i>seln_demo</i>	128
Large Selections	128

The Selection Service and Library

9.1. Introduction

The Selection Service package provides for flexible communication among window applications. It is concerned with aspects of the selection[s] the user has made, and with the status of the user interface which may affect those selections. It has 3 distinct aspects:

1. A server process maintains a clearinghouse of information about the selection, and the function keys which may affect how a selection is made. This process responds to RPC requests for information from clients. Normally, the RPC accesses will be done only by library routines described below; therefore details of that access do not appear in this manual.
2. A library of client routines is provided to communicate with the clearinghouse process and with each other. These routines allow a client to acquire a selection, or yield it to another application, to determine the current holder of a selection, and send or receive requests concerning a selection's contents and attributes.
3. A minimal set of requests is defined for communicating between applications which have some interest in the selection. This set is deliberately separated from the transport mechanism mentioned under (2) above, and the form of a request is carefully separated from its content. This allows applications to treat the definition of what can be said about the selection as open-ended; anything consenting applications agree to can be passed through the Selection Service.

This chapter is primarily concerned with the transport library, and how to use that protocol to accomplish representative application tasks. The (current) set of generic requests is also presented, and used in illustrations.

The next sections is a fast overview of how the Selection Service works, and a walk-through of the complex protocol followed when selections are transferred between client. This is followed by several discursive sections devoted to particular aspects of using the Selection Service, such as reporting function-key transitions, sending requests, acquiring and releasing selections, replying to requests, and debugging selection applications. Throughout these sections, some procedures and data types are mentioned or described. Full documentation for all of these may be found in the reference section which follows. Finally there are two example programs which show how to use the Selection Service for simple queries (*get_selection*) and how a real client of the selection service works (*seln_demo*).

The remainder of the chapter comprises reference material: descriptions of the public data and procedures of the selection library, a list of the defined common attributes, and the complete source of a program to retrieve the contents of a selection and print it on *stdout*.

9.2. Basic Concepts

When a user makes a selection, it is some application program which interprets the mouse and function-key events and resolves them into a particular selection. The Selection Service is involved only as the processing of function-keys and selections spans application windows. Application programs interact with the package in proportion to the sophistication of their requirements. This section is intended to present the information necessary for any use of the Selection Service, and to indicate what further information in the document pertains to various uses of the package.

The selection library deals with four objects under the general term “*selection*.” Most familiar is the *Primary* selection, which is normally indicated on-screen by inverting (“highlighting”) its contents. Selections made while a function key is held down (usually indicated with an underscore) are *Secondary* selections. The selection library treats the *Shelf*²¹ (the global buffer which is loaded by *Cut*²² and *Copy* operations, and which may be retrieved by a *Paste* operation) as a third kind of selection. Finally, the insertion point, or *Caret*, is also treated as a selection, even though it has no contents. These are the four *ranks* the selection library deals with: *Caret*, *Primary*, *Secondary*, and *Shelf*.

Every selection has a *holder*; this is a piece of code within a process which is responsible for operating on the selection and responding to other applications' requests about it. A selection holder is a *client* of the selection library. Typically, a selection client is something like a subwindow; there may be several selection clients within a single process.

Because the selection library uses RPC as well as the SunView input system, it relies on the SunView Notifier to dispatch events; thus any application using the selection library must be built on the notifier system.

9.3. Fast Overview

The simplest use of the Selection Service is to inquire about a selection held by some other application. Programs which never make a selection will not use the facilities described in the rest of this section. Much of the material remaining before the beginning of reference section is likewise irrelevant to these programs: the sections on *Acquiring and Releasing Selections* and *Callback Procedures* pertain only to clients which make selections.

A program which will actually make selections should be a full-fledged client of the selection library. Such an application calls `seln_create()` during the client's initialization routines; if successful, this returns an opaque client handle which is then passed back in subsequent calls to selection library procedures.

²¹ The shelf is called the *Clipboard* in user-oriented documentation such as the *SunView 1 Beginner's Guide*, and in the text subwindow menu.

²² Prior to SunOS Release 4.0, the terms *Delete*, *Put*, and *Get* were used instead of the “industry-standard” *Cut*, *Copy*, and *Paste*.

The Selection Service vs. the Selection Library

Two arguments to this create-procedure specify client *callback procedures* which may be called to perform processing required by external events. These are the `function_proc()` and the `reply_proc()` described below.

Note the difference here between the Selection Service program, and the selection library. The former is a separate program, `selection_svc(1)`, the latter is the code in a SunView application which knows how to talk to the Selection Service and its other clients. After a client is successfully created, it may call library routines to:

- inquire of the Selection Service how to get in touch with the holder of a particular rank of selection,
- send a request to such a holder (e.g. to find out the contents of a selection),
- inform the Selection Service of any change in the state of the function keys
- acquire a selection, and later,
- release a selection it has acquired.

The client must be prepared to deal with:

- reports from the Selection Service of function key transitions it may be interested in (these are handled by its `function_proc()`)
- requests from other clients about selections it holds (these are handled by its `reply_proc()`).

Finally, when the client is finished, it calls `seln_destroy()` to clean up its selection library resources.

9.4. Selection Service Protocol: an Example

It's important to remember that the Selection Service is an external process which client programs talk to to find out about the selections held by other clients. To clarify the conceptual model, here's an outline of the communication which takes place when a user makes a secondary selection between two windows.

Secondary Selection Between Two Windows

Try the sequence out yourself while you read the description; The following description assumes that you have a standard set-up (focus follows cursor, adjusting selections does not make them pending-delete, left mouse button is select, middle is extend/adjust, etc.). If you have trouble performing the operation, the *SunView 1 Beginner's Guide* has more to say on selections.

1. Move the cursor into one text subwindow (call it "window A"). Note how the border of the subwindow darkens.
2. Click LEFT in window A to select a character. Window A sets the caret and highlights the primary selection you have just made.
3. Hold down the `[Paste]` key (usually `[L8]`).
4. Move the cursor into another text subwindow (call it "window B").
5. Click LEFT and MIDDLE in window B to select some text. Note how it underlines the selection to indicate that it is a secondary selection.

6. Release the **(Paste)** key. Note how the secondary selection from window B appears at the caret in window A.

Here's what happens internally when this sequence of user actions takes place. Assume window A and B have registered as clients of the Selection Service using `seln_create()`. The initial state is no function keys down, and no one holds any of the selections.

Table 9-1 *Selection Service Scenario*

<i>Client A</i>	<i>Selection Service</i>	<i>Client B</i>	<i>Notes</i>
<i>User moves cursor into window A.</i> Get KBD_USE event. Call <code>seln_acquire(A,</code> <code>SELN_CARET)</code> to acquire caret. Provide feedback to indi- cate window has keyboard focus. Done automatically by the window packages.	Note caller as holder of Caret; in this case assume no previous holder, so no notification.		1
<i>User presses left mouse button (but has not released yet).</i> <code>seln_inquire(</code> <code>UNSPECIFIED)</code> Provide feedback to indicate selection at indicated point.	Check state of function keys; since all are up, return primary.		2
<i>User releases left mouse button.</i> <code>seln_acquire(A,</code> <code>SELN_PRIMARY)</code>	Note caller as holder of primary.		
<i>User holds [Paste] key down (but has not released yet).</i> <code>seln_report_event(A,</code> <code>down-event)</code>	Note (Paste) key is down.		
<i>User moves cursor into window B.</i>		Get KBD_USE event.	

Table 9-1 Selection Service Scenario— Continued

<i>Client A</i>	<i>Selection Service</i>	<i>Client B</i>	<i>Notes</i>
<i>User presses left mouse button (but has not released yet).</i>	<p>Check state of function keys; since [Paste] is Down, note caller as secondary holder; return secondary.</p>	<pre>selsn_acquire(UNSPECIFIED)</pre> <p>Provide feedback to indicate secondary selection at indicated point. If user has held down the [Control] key, secondary selection may be pending-delete.</p>	3
<i>User releases left mouse button.</i>		no action required	
<i>User presses middle mouse button (but has not released yet).</i>		provide feedback for adjustment of secondary selection	
<i>User releases middle mouse button.</i>		no action required	
<i>User releases [Paste] key.</i>	<p>Note that this key-up leaves all function keys up. Figure out which of the four selection holders to notify (the primary selection holder, A).</p>	<pre>selsn_report_event(B, up-event)</pre> <p>B's call to <code>selsn_report_event()</code> gets notification in return; B calls <code>selsn_figure_response()</code>, which says IGNORE</p>	

Table 9-1 Selection Service Scenario—Continued

<i>Client A</i>	<i>Selection Service</i>	<i>Client B</i>	<i>Notes</i>
<p>Window A's <code>function_proc()</code> is called (it was registered with the Selection Service in <code>seln_create()</code> to handle reported function key transitions). The <code>function_proc()</code> in turn calls <code>seln_figure_response()</code>, which tells it to request the contents of the secondary selection from the secondary selection holder.</p> <p>So <code>function_proc()</code> calls <code>seln_init_request()</code> to set up a request to</p> <ul style="list-style-type: none"> a) get the contents of the secondary selection, b) commit any pending-delete, and yield secondary; <p>It then calls <code>seln_request()</code> to send this request to B.</p>			
			<p>The request arrives as an RPC call to B's <code>request-proc()</code> (likewise registered with the Selection Service in <code>seln_create()</code>, but the reply proc handles requests about selections). B provides response to each attribute in the request; in response to the request to <code>Yield(secondary)</code>, it calls <code>seln_done(SELN_SECONDARY)</code></p>
	Note there is no holder of the secondary selection.		
			<p>Do pending delete if necessary: remove selection; cache contents; call <code>seln_acquire(SELN_SHELF)</code></p>
	Note caller as holder of the Shelf.		
			Return from rpc call.

Table 9-1 *Selection Service Scenario—Continued*

<i>Client A</i>	<i>Selection Service</i>	<i>Client B</i>	<i>Notes</i>
Consume response, inserting contents; if primary selection were pending delete, then would delete primary (superseding B as holder of Shelf if so).			

Notes on the Above Example

1. To be absolutely correct, clients should determine whether they want the Caret when they get a KBD_USE event, and acquire it immediately if so. Code to do this would look something like

<i>Client A</i>	<i>Selection Service</i>	<i>Client B</i>	<i>Notes</i>
<i>User moves cursor into window A.</i> Get KBD_USE event. Call <code>seln_get_function_state()</code> <code>seln_acquire(SELN_CARET)</code>	Return state of function keys (all up).		

But this would involve waking up a client every time the cursor crossed its window; further, the kernel sends spurious KBD_USEs on window-crossings when a function key is down; Hence most clients postpone acquiring the Caret until they know they have input. For most clients, input is usually signalled by a key going down (either a function-key or a regular ASCII-key).

2. Some window types acquire the Caret and primary selection at this point, so they simply call `seln_acquire(client_handle, SELN_UNSPECIFIED)` when a key goes down.
3. This "Point Down" describes what most windows implement, as opposed to the first theoretical exchange. The issue is whether clients call `seln_inquire(client, SELN_UNSPECIFIED)`, and then acquire the appropriate selection, or whether they should simply call `seln_acquire(client, SELN_UNSPECIFIED)`.

9.5. Topics in Selection Processing

Reporting Function-Key Transitions

When an application makes a selection, its rank depends on the state of the function keys. (A secondary selection is one made while a function key is held down.) The application which is affected by a function-key transition may not be the one whose window received that transition. Consequently, this system requires that the service be informed of transitions on those function keys that affect the rank of a selection; the service then provides that state information to

any application which inquires.

The keys which affect the rank of a selection are **Copy**, **Paste**, **Cut**, and **Find** (ordinarily these are keys **L6**, **L8**, **L10**, and **L9**, respectively). If an application program does not include these events in its input mask, then they will fall through to the root window, and be reported by it. But if the application is reading these events for any reason, then it should also report the event to the service. `seln_report_event()` is the most convenient procedure for this purpose; `seln_inform()` does the work at a lower level.

When the service is told a function key has gone up, it will cause calls to the *function_proc* callback procedures of the holders of each selection. For the client that reports the key-up, this will happen during the call to `seln_report_event`; for other holders, it can happen any time control returns to the client's notifier code. The required processing is detailed under *Callback Procedures* below. Programs which never called `seln_create()` can call `seln_report_event()` without incurring any extra processing — they have no *function_proc* to call.)

Two procedures are provided so that clients may interrogate state of the functions keys as stored by the service: `seln_get_function_state()` takes a `Seln_function` and returns TRUE or FALSE as the service believes that function key is down or not. `seln_functions_state()` takes a pointer to a `Seln_functions_state` buffer (a bit array which will be all 0 if the service believes all function keys are currently up).

Sending Requests to Selection Holders

Clients of the Selection Service inquire of it who holds a particular rank of selection using `seln_inquire()`; they then talk directly to each other through request buffers. A client loads an attribute value list of its requests about the rank of selection into these buffers. Requests include things like "What are the contents of the selection?", "What line does the selection start on?", "Please give up the selection (I need to highlight)!", and so forth.

Inside the selection library, a *request* is a buffer (a `Seln_request` structure); the following declarations are relevant to the processing that is done with such a buffer:

```
typedef struct {
    Seln_result      (*consume)();
    char             *context;
} Seln_requester;

typedef struct {
    Seln_replier_data *replier;
    Seln_requester    requester;
    char              *addressee;
    Seln_rank          rank;
    Seln_result        status;
    unsigned           buf_size;
    char               data[SELN_BUFSIZE];
} Seln_request;
```

```

/* VARARGS */
Seln_request *
seln_ask(holder, <attributes>, ... 0)
    Seln_holder      *holder;
    Attr_union       attribute;

/* VARARGS */
void
seln_init_request(buffer, holder, <attributes>, ... 0)
    Seln_request      *buffer;
    Seln_holder       *holder;
    char              *attributes;

/* VARARGS */
Seln_result
seln_query(holder, reader, context, <attributes>, ... 0)
    Seln_holder      *holder;
    Seln_result       (*reader)();
    char              *context;
    Attr_union       attribute;

Seln_result
reader(buffer)
    Seln_request      *buffer;

```

The selection library routines pass request buffers from clients to the particular client who holds that rank of selection. That client returns its reply in one or more similar buffers. The library is responsible for maintaining the dialogue, but does not have any particular understanding of the requests or their responses.

`seln_query()` (or `seln_ask()`, for clients unwilling to handle replies of more than one buffer) is used to construct a request and send it to the holder of a selection.

There is a lower-level procedure, `seln_request()` which is used to send a pre-constructed buffer containing a request, and an initializer procedure, `seln_init_request()` which can be used to initialize such a buffer.

The data portion of the request buffer is an attribute-value list, copied from the `<attributes> ...` arguments in a call to `seln_query()` or `seln_ask()`. A similar list is returned in the reply, typically with real values replacing placeholders provided by the requester. (It may take several buffers to hold the whole reply list; this case is discussed below.)

The request mechanism is quite general: an attribute-value pair in the request may indicate some action the holder of the selection is requested to perform — even a program to be executed may be passed, as long as the requester and the holder agree on the interpretation.

The header file `<suntool/selection_attributes.h>` defines a base set of request attributes; a copy is printed near the end of this chapter. The most common request attribute is `SELN_REQ_CONTENTS_ASCII`, which requests

the holder of the selection to return its contents as an ASCII string.

Long Request Replies

If the reply to a request is very long (more than about 2000 bytes), more than one buffer will be used to return the response. In this case, `seln_ask()` simply returns a pointer to the first buffer and discards the rest.

NOTE *This means that the attribute value list in the first buffer may not be properly terminated (but it probably will be).*

`seln_query()` should be used if long replies are to be handled gracefully. Rather than returning a buffer, it repeatedly calls a client procedure to handle each buffer in turn. The client passes a pointer to the procedure to be called in the `reader` argument of the call to `seln_query()` (that address appears in the `consume` element of the `Seln_requester` struct.) Such procedures typically need some context information to be saved across their invocations; this is provided for in the `context` element of the `Seln_requester` structure. This is a 32-bit datum provided for the convenience of the `reader` procedure; it may be filled in with literal data or a pointer to some persistent storage; the value will be available in each call to `reader`, and may be modified at will.

Selection holders are responsible for processing and responding to the attributes of a request in the order they appear in the request buffer. Selection holders may not recognize all the attributes in a request; there is a standard response for this case: In place of the unrecognized attribute (and its value, if any), the replier inserts the attribute `SELN_REQ_UNRECOGNIZED`, followed by the original (unrecognized) attribute. This allows heterogeneous applications to negotiate the level at which they will communicate.

A straightforward example of request processing (including code to handle a long reply) is the `get_selection` program, which appears at the end of this chapter.

Acquiring and Releasing Selections

Applications in which a selection can be made must be able to tell the service they now hold the selection, and they must be able to release a selection, either on their own initiative, or because another application has asked to acquire it. `Seln_acquire` is used both to request a current holder of the selection to yield, and then to inform the service that the caller now holds that selection. `seln_yield()` is used to yield the selection on the caller's initiative. A request to yield because another application is becoming the holder is handled like other requests; this is discussed under *Callback Procedures* below.

Callback Procedures: Function-Key Notifications

When you register a client with the Selection Service using `seln_create()`, you must supply a *function_proc()* and a *reply_proc()*. The former is called by the Selection Service to report function-key transitions (which may have occurred in other windows) to the client; the latter is called by the Selection Service when this client holds one of the selections and other clients have requests about it.

The selection library calls the client's *function_proc()* when the Selection Service is informed of a function-key transition which leaves all function keys up. This may happen inside the client's call to `seln_report_event()`, if the reporting client holds a selection; otherwise the call will arrive through the RPC

mechanism to the `function_proc()`.

The relevant declarations are:

```
typedef enum {
    SELN_IGNORE, SELN_REQUEST, SELN_FIND,
    SELN_SHELVE, SELN_DELETE
} Seln_response;

typedef struct {
    Seln_function      function;
    Seln_rank          addressee_rank;
    Seln_holder        caret;
    Seln_holder        primary;
    Seln_holder        secondary;
    Seln_holder        shelf;
} Seln_function_buffer;

Seln_response
seln_figure_response(buffer, holder)
    Seln_function_buffer *buffer;
    Seln_holder          **holder;

void
function_proc(client_data, function)
    char          *client_data;
    Seln_function_buffer *function;
```

`function_proc()` will be called with a copy of the 32 bits of client data originally given as the third argument to `seln_create()`, and a pointer to a `Seln_function_buffer`. The buffer indicates what function is being invoked, which selection the called program is expected to be handling, and what the Selection Service knows about the holders of all four selection ranks (one of whom is the called program). A client will only be called once, even if it holds more than one selection. (In that case, the buffer's `addressee_rank` will contain the first rank the client holds.)

Responding to Selection Requests

The holders of the selections are responsible for coordinating any data transfer and selection-relinquishing among themselves. The procedure `seln_figure_response()` is provided to assist in this task. It takes a pointer to a function buffer such as the second argument to a `function_proc` callback, and a pointer to a pointer to a `Seln_holder`. It returns an indication of the action which this client should take according to the standard interface. It also changes the `addressee_rank` element of that buffer to be the rank which is affected (the destination of a transfer, the item to be deleted, etc.), and if interaction with another holder is required, it stores a pointer to the appropriate `Seln_holder` element in the buffer into the location addressed by the second argument. Here are the details for each return value:

SELN_IGNORE	No action is required of this client. Another client may make a request concerning the selection(s) this client holds.
-------------	--

- SELN_REQUEST** This client is expected to request the contents of another selection and insert them in the location indicated by `buffer->addressee_rank`. The holder of the selection that should be retrieved is identified by `*holder`. If `*holder` points to `buffer->secondary`, the request should include `SELN_REQ_YIELD`; if it points to `buffer->primary` or `buffer->secondary`, the request should include `SELN_REQ_COMMIT_PENDING_DELETE`.
- Example: the called program holds the Caret and Primary selection; the **Paste** key went up, and there is no Secondary selection. The return value will be `SELN_REQUEST`, `buffer->addressee_rank` will be `SELN_CARET` and `*holder` will be the address of `buffer->shelf`. The client should request the contents of the shelf from that holder.
- SELN_FIND** This client should do a **Find** (if it can). `Buffer->addressee_rank` will be `SELN_CARET`; if `*holder` is not NULL, the target of the search is the indicated selection. If `*holder` points to `buffer->secondary`, the request should include `SELN_REQ_YIELD`.
- SELN_SHELV** This client should acquire the shelf from `*holder` (if that is not NULL), and make the current contents of the primary selection (which it holds) be the contents of the shelf.
- SELN_DELETE** This client should delete the contents of the secondary selection if it exists, or else the primary selection, storing those contents on the shelf. `Buffer->addressee_rank` indicates the selection to be deleted; `*holder` indicates the current holder of the shelf, who should be asked to yield.
- `Seln_secondary_exists` and `seln_secondary_made` are predicates which may be of use to an application which is not using `seln_figure_response()`. Each takes a `Seln_function_buffer` and returns TRUE or FALSE. When the user has made a secondary selection and then cancelled it, `seln_secondary_made` will yield TRUE while `seln_secondary_exists` will yield FALSE. This indicates the function-key action should be ignored.

Callback Procedures: Replying to Requests

The client's `reply_proc()` callback procedure is called when another application makes a request concerning a selection held by this client. It is invoked once for each attribute in the request, plus once for a terminating attribute supplied by the selection library. The relevant declarations are:

```

typedef struct {
    char          *client_data;
    Seln_rank     rank;
    char          *context;
    char          **request_pointer;
    char          **response_pointer;
} Seln_replier_data;

Seln_result
reply_proc(item, context, length)
    caddr_t      item;
    Seln_replier_data *context;
    int          length;

```

`reply_proc()` will be called with each of the attributes of the request in turn. `item` is the attribute to be responded to; `context` points to data which may be needed to compute the response, and `length` is the number of bytes remaining in the buffer for the response. `reply_proc()` should write any appropriate response/value for the given attribute into the buffer indicated in `context->response_pointer`, and return.

The fields of `*context` contain, in order:

- the 32 bits of private client data passed as the last argument to `seln_create()`, returned for the client's convenience;
- the rank of the selection this request is concerned with;
- a holder for 32 more bits of context for the replier's convenience (this will typically hold a pointer to data which allows a client to maintain state while generating a multi-buffer response);
- a pointer to a pointer into the request buffer, just after the current item (so that the replier may read the value of this item if relevant). *This pointer should not be modified by `reply_proc`.*
- a pointer to a pointer into the response buffer, where the value / response (if any) for this item should be stored. This pointer should be updated to point past the end of the response stored. (Note that items and responses should always be multiples of full-words; thus, this pointer should be left at an address which is 0 mod 4.)

After storing the response to one item, `reply_proc` should return `SELN_SUCCESS` and await the next call. When all attributes in a request have been responded to, `reply_proc` will be called one more time with `item == SELN_REQ_END_REQUEST`, to give it a chance to clean up any internal state associated with the request.

Two attributes which are quite likely to be encountered in the processing of a request due to a function-key event, `SELN_REQ_COMMIT_PENDING_DELETE` and `SELN_REQ_YIELD`, are concerned more with the proper handling of secondary selections (rather than the needs of the requesting application), so they are discussed here.

`SELN_REQ_COMMIT_PENDING_DELETE` indicates that a secondary selection which was made in pending-delete mode should now be deleted. If the recipient does not hold the secondary selection, or the secondary selection is not in pending-delete mode, the replier should ignore the request, i.e., simply return `SELN_SUCCESS` and await the next call. `SELN_REQ_YIELD`, with an argument of `SELN_SECONDARY`, means the secondary selection should be deselected, if it still exists.

Complications on this basic model will now be addressed in order of increasing complexity.

If the request concerns a selection the application does not currently hold, `reply_proc` should return `SELN_DIDNT_HAVE` immediately; it will not be called further for that request.

If the request contains an item the client isn't prepared to deal with, `reply_proc` should return `SELN_UNRECOGNIZED` immediately; the selection library will take care of manipulating the response buffer to have the standard unrecognized-format, and call back to `reply_proc` with the next item in the list.

Finally, a response to a request may be larger than the space remaining in the buffer — or larger than several buffers, for that matter. This situation will never arise on items whose response is a single word — the selection library ensures there is room for at least one 4-byte response in the buffer before calling `reply_proc`.

If a response is too big for the current buffer, the replier should store as much as fits in `length` bytes, save sufficient information to pick up where it left off in some persistent location, store the address of that information in `context->context`, and return `SELN_CONTINUED`. Note that the replier's context information should not be local to `reply_proc`, since that procedure will exit and be called again before the information is needed.

The selection library will ship the filled buffer to the requester, and prepare a new one for the continuation of the response. It will then call `reply_proc` again, with the same item and context, and `length` indicating the space available in the new buffer. `reply_proc()` should be able to determine from `context->context` that it has already started this response, and where to continue from. It continues by storing as much of the remainder of the response as fits into the buffer, updating `context->response_pointer` (and its own context information), and again returning `SELN_CONTINUED` if the response is not completed. When the end of the response has been stored, including any terminator if one is required, the private context information may be freed, and `reply_proc` should return `SELN_SUCCESS`.

The next call to `reply_proc` will be to respond to the next item in the request if there is one, or else to `SELN_REQ_END_REQUEST`.

9.6. Debugging and Administrative Facilities

A number of aids to debugging have been included in the system for applications which use the Selection Service. In addition to providing information on how to access holders of selections and maintaining the state of the user-interface keys, the service will respond to requests to display traces of these requests, and to dump its internal state on an output stream. `Seln_debug` is used to turn service tracing on or off; `seln_dump` instructs the service to dump all or part of its state on `stderr`.

A number of library procedures provide formatted dumps of Selection Service structs and enumerated types. These can be found below as `seln_dump_*`.

In debugging an application which uses the Selection Service, it may be convenient to use a separate version of the service whose state is affected only by the application under test. This is done by starting the service with the `-d` flag; that is, by entering `/usr/bin/selection_svc -d &` to a shell. The resulting service will use a different RPC program number from the standard version, but be otherwise identical. The two versions of the service may be running at the same time, each responding to its own clients. A client may elect (via `seln_use_test_service()`) to talk to the test service. Thus, it is easy to arrange to have an application under development talking to its own service, while running under a debugger which is talking to a standard service — this keeps the debugger, editors, etc. from interfering with the state maintained by the test service.

The Selection Service depends heavily on remote procedure calls, using Sun's RPC library. It is always possible that the called program has terminated or is not responding for some other reason; this is often detected by a timeout. The standard timeout at this writing is 10 seconds; this is a compromise between allowing for legitimate delays on loaded systems, and minimizing lockups when the called program really won't respond. The delay may be adjusted by a call to `seln_use_timeout`.

9.7. Other Suggestions

Always call `seln_figure_response()` to determine what to do with a request.

Use `seln_report_event()` instead of `seln_inform()` to report events you aren't interested in. Note that the canvas subwindow by default does *not* report SunView function-key transitions; it relies on the events falling through to the frame or root window, which does report the transitions.

If you expect to **Paste** the selection, you must acquire the Caret beforehand.

If You Die

If your application dies, if it holds the shelf then you should save its contents by writing them to a file and calling `seln_hold_file()`. You should also yield the primary and secondary selections.

9.8. Reference Section

The reference material which follows presents first the header files and the public data definitions they contain; then it lists each public procedure in the selection library (in alphabetical order) with its formal parameter declarations, return value, and a brief description of its effect.

Required Header Files

```
#include <sunwindow/attr.h>
#include <suntool/selection_svc.h>
#include <suntool/selection_attributes.h>
```

Enumerated Types

```
typedef enum {
    SELN_FAILED,      SELN_SUCCESS,      /* basic uses */
    SELN_NON_EXIST,   SELN_DIDNT_HAVE,   /* special cases */
    SELN_WRONG_RANK,  SELN_CONTINUED,
    SELN_CANCEL,      SELN_UNRECOGNIZED
} Seln_result;

typedef enum {
    SELN_UNKNOWN, SELN_CARET, SELN_PRIMARY,
    SELN_SECONDARY, SELN_SHELF, SELN_UNSPECIFIED
} Seln_rank;

typedef enum {
    SELN_FN_ERROR,
    SELN_FN_STOP, SELN_FN_AGAIN,
    SELN_FN_PROPS, SELN_FN_UNDO,
    SELN_FN_FRONT, SELN_FN_PUT,
    SELN_FN_OPEN, SELN_FN_GET,
    SELN_FN_FIND, SELN_FN_DELETE
} Seln_function;

typedef enum {
    SELN_NONE, SELN_EXISTS, SELN_FILE
} Seln_state;

typedef enum {
    SELN_IGNORE, SELN_REQUEST, SELN_FIND,
    SELN_SHELVE, SELN_DELETE
} Seln_response;
```

Other Data Definitions

```
typedef char *Seln_client;

typedef struct {
    Seln_rank      rank;
    Seln_state     state;
    Seln_access    access;
} Seln_holder;

Seln_holder
```

```

typedef struct {
    Seln_holder    caret;
    Seln_holder    primary;
    Seln_holder    secondary;
    Seln_holder    shelf;
}    Seln_holders_all;

typedef struct {
    Seln_function  function;
    Seln_rank      addressee_rank;
    Seln_holder    caret;
    Seln_holder    primary;
    Seln_holder    secondary;
    Seln_holder    shelf;
}    Seln_function_buffer;

typedef struct {
    char            *client_data;
    Seln_rank       rank;
    char            *context;
    char            **request_pointer;
    char            **response_pointer;
}    Seln_replier_data;

typedef struct {
    Seln_result      (*consume)();
    char            *context;
}    Seln_requester;

#define SELN_BUFSIZE
    (1500 - sizeof(Seln_replier_data *)
    - sizeof(Seln_requester)
    - sizeof(char *)
    - sizeof(Seln_rank)
    - sizeof(Seln_result)
    - sizeof(unsigned))

typedef struct {
    Seln_replier_data *replier;
    Seln_requester     requester;
    char               *addressee;
    Seln_rank           rank;
    Seln_result         status;
    unsigned            buf_size;
    char                data[SELN_BUFSIZE];
}    Seln_request;

```


Procedure Declarations

```
Seln_rank
seln_acquire(client, asked)
    Seln_client  client;
    Seln_rank    asked;
```

`client` is the opaque handle returned from `seln_create()`; the client uses this call to become the new holder of the selection of rank `asked`. `asked` should be one of `SELN_CARET`, `SELN_PRIMARY`, `SELN_SECONDARY`, or `SELN_SHELF`, `SELN_UNSPECIFIED`. If successful, the rank actually acquired is returned.

If `asked` is `SELN_UNSPECIFIED`, the client indicates it wants whichever of the primary or secondary selections is appropriate given the current state of the function keys; the one acquired can be determined from the return value.

```
/* VARARGS */
Seln_request *
seln_ask(holder, <attributes>, ... 0)
    Seln_holder      *holder;
    Attr_union       attribute;
```

`seln_ask()` looks and acts very much like `seln_query()`; the only difference is that it does not use a callback proc, and so cannot handle replies that require more than a single buffer. If it receives such a long reply, it returns the first buffer, and discards all that follow. The return value is a pointer to a static buffer; in case of error, this will be a valid pointer to a null buffer (`buffer->status = SELN_FAILED`). `seln_ask()` is provided as a slightly simpler interface for applications that refuse to process long replies.

```
void
seln_clear_functions()
```

The Selection Service is told to forget about any function keys it thinks are down, resetting its state to all-up. If it knows of a current secondary selection, the service will tell its holder to yield.

```
Seln_client
seln_create(function_proc, reply_proc, client_data)
    void      (*function_proc) ();
    void      (*reply_proc) ();
    caddr_t    client_data);
```

The selection library is initialized for this client. If this is the first client in its process, an RPC socket is established and the notifier set to recognize incoming calls. `Client_data` is a 32-bit opaque client value which the Selection Service will pass back in callback procs, as described above. The first two arguments are addresses of client procedures which will be called from the selection library when client processing is required. These occasions are:

- when the service sees a function-key transition which may interest this client, and
- when another process wishes to make a request concerning the selection this client holds,

Details of these procedures are described under *Callback Procs*, above.

```
/* VARARGS */
Seln_result
seln_debug(<attributes> ... 0)
    Seln_attribute  attribute;
```

A debugging routine which requests the service to turn tracing on or off for specified calls. Each attribute identifies a particular call; its value should be 1 if that call is to be traced, 0 if tracing is to be stopped. The attributes are listed with other request-attributes in the first appendix. Tracing is initially off for all calls. When tracing is on, the Selection Service process prints a message on its `stderr` (typically the console) when it enters and leaves the indicated routine.

```
void
seln_destroy(client)
    Seln_client  client;
```

A client created by `seln_create` is destroyed: any selection it may hold is released and various pieces of data associated with the selection mechanism are freed. If this is the last client in this process using the Selection Service the RPC socket is closed and its notification removed.

```
Seln_result
seln_done(client, rank)
    Seln_client  client;
    Seln_rank    rank;
```

Client indicates it is no longer the holder of the selection of the indicated rank. The only cause of failure is absence of the Selection Service. It is not necessary for a client to call this procedure when it has been asked by another client to yield a selection; the service will be completely updated by the acquiring client.

```
void
seln_dump_function_buffer(stream, ptr)
    FILE          *stream;
    Seln_function_buffer *ptr;
```

A debugging routine which prints a formatted display of a `Seln_function_buffer` struct on the indicated stream.


```
void
seln_dump_function_key(stream, ptr)
    FILE          *stream;
    Seln_function *ptr;
```

A debugging routine which prints a formatted display of a Seln_function_key transition on the indicated stream.

```
void
seln_dump_holder(stream, ptr)
    FILE          *stream;
    Seln_holder   *ptr;
```

A debugging routine which prints a formatted display of a Seln_holder struct on the indicated stream.

```
void
seln_dump_rank(stream, ptr)
    FILE          *stream;
    Seln_rank     *ptr;
```

A debugging routine which prints a formatted display of a Seln_rank value on the indicated stream.

```
void
seln_dump_response(stream, ptr)
    FILE          *stream;
    Seln_response *ptr;
```

A debugging routine which prints a formatted display of a Seln_response value on the indicated stream.

```
void
seln_dump_result(stream, ptr)
    FILE          *stream;
    Seln_result   *ptr;
```

A debugging routine which prints a formatted display of a Seln_result value on the indicated stream.

```
void
seln_dump_service(rank)
    Seln_rank  rank;
```

A debugging routine which requests the service to print a formatted display of its internal state on its standard error stream. Rank determines which selection holder is to be dumped; if it is SELN_UNSPECIFIED, all four are printed. In any case, the dump concludes with the state of the function keys and the set of

open file descriptors in the service.

```
void
seln_dump_state(stream, ptr)
    FILE          *stream;
    Seln_state     *ptr;
```

A debugging routine which prints a formatted display of a `Seln_state` value on the indicated stream.

```
Seln_response
seln_figure_response(buffer, holder)
    Seln_function_buffer *buffer;
    Seln_holder          **holder;
```

A procedure to determine the correct response according to the standard user interface when `seln_inform()` returns `*buffer`, or the client's *function_proc* is called with it. The field `addressee_rank` will be modified to indicate the selection which should be affected by this client; `holder` will be set to point to the element of `*buffer` which should be contacted in the ensuing action, and the return value indicates what that action should be.

```
Seln_result
seln_functions_state(buffer)
    Seln_functions_state *buffer;
```

The service is requested to dump the state it is maintaining for the function keys into the bit array addressed by `buffer`. At present, the only commitment made to representation in the buffer is that some bit will be on (`== 1`) for each function key which is currently down. Thus this call can be used to determine whether any function keys are down, but not which. `SELN_SUCCESS` is returned unless the service could not be contacted.

```
int
seln_get_function_state(which)
    Seln_function    which;
```

A predicate which returns `TRUE` when the service's state shows the function key indicated by `which` is down and `FALSE` otherwise.

```
Seln_result
seln_hold_file(rank, path)
    Seln_rank    rank;
    char         *path;
```

The Selection Service is requested to act as the holder of the specified `rank`, whose ASCII contents have been written to the file indicated by `path`. This allows a selection to persist longer than the application which made it can

maintain it. Most commonly, this will be done by a process which holds the shelf when it is about to terminate.

```
int
seln_holder_same_client(holder, client_data)
    Seln_holder      *holder;
    char             *client_data;
```

A predicate which returns TRUE if the holder referred to by `holder` is the same selection client as the one which provided `client_data` as its last argument to `seln_create`.

```
int
seln_holder_same_process(holder)
    Seln_holder *holder;
```

A predicate which returns TRUE if the holder is a selection client in the same process as the caller. (This procedure is used to short-circuit RPC calls with direct calls in the same address space.)

```
Seln_function_buffer
seln_inform(client, which, down)
    Seln_client      client;
    Seln_function     which;
    int              down;
```

This is the low-level, policy-independent procedure for informing the Selection Service that a function key has changed state. Most clients will prefer to use the higher-level procedure `seln_report_event()`, which handles much of the standard interpretation required.

`Client` is the client handle returned from `seln_create()`; it may be 0 if the client guarantees it will never need to respond to the function transition. `Which` is an element of the `Seln_function` enum defined in `<suntool/selection_svc.h>`; `down` is a boolean which is TRUE if the key went down.

On an up-event which leaves all keys up, the service informs the holders of all selections of the transition, and what other parties are affected. If the caller of `seln_inform()` is one of these holders, its notification is returned as the value of the function; other notifications go out as a call on the client's *function_proc* callback procedure (described above under *Callback Procedures*). Only one notification is sent to any single client. If the caller does not hold any selection, or if the transition was not an up which left all function keys up, the return value will be a null `Seln_function_buffer`; `buffer.rank` will be `SELN_UNKNOWN`.

```

/* VARARGS */
void
seln_init_request(buffer, holder, <attributes>, ... 0)
    Seln_request      *buffer;
    Seln_holder       *holder;
    char              *attributes;

```

This procedure is used to initialize a buffer before calling `seln_request`. (It is also called internally by `seln_ask` and `seln_query`.) It takes a pointer to a request buffer, a pointer to a struct referring to the selection holder to which the request is to be addressed, and a list of attributes which constitute the request to be sent. The attributes are copied into `buffer->data`, and the corresponding size is stored into `buffer->buf_size`. Both elements of `requester_data` are zeroed; if the caller wants to handle long requests, a *consumer-proc* and *context* pointers must be entered in these elements after `seln_init_request` returns.

```

Seln_holder
seln_inquire(rank)
    Seln_rank  rank;

```

A `Seln_holder` struct is returned, containing information which enables the holder of the indicated selection to be contacted. If the `rank` argument is `SELN_UNSPECIFIED`, the Selection Service will return access information for either the primary or secondary selection holder, as warranted by the state of the function keys it knows about; the `rank` element in the returned struct will indicate which is being returned.

This procedure may be called without having called `seln_create()` first. If no contact between this process and the service has been established yet, it will be set up, and then the call will proceed as usual. In this case, return of a null holder struct may indicate inaccessibility of the server.

```

Seln_holders_all
seln_inquire_all()

```

A `Seln_holders_all` struct is returned from the Selection Service; it consists of a `Seln_holder` struct for each of the four ranks.


```

Seln_result
reader(buffer)
    Seln_request      *buffer;

/* VARARGS */
Seln_result
seln_query(holder, reader, context, <attributes>, ... 0)
    Seln_holder      *holder;
    Seln_result      (*reader) ();
    char             *context;
    Attr_union       attribute;

```

A request is transmitted to the selection holder indicated by the holder argument. Consume and context are used to interpret the response, and are described in the next paragraph. The remainder of the arguments to `seln_query` constitute an attribute value list which is the request. (The last argument should be a 0 to terminate the list.)

The procedure pointed to by consume will be called repeatedly with a pointer to each buffer of the reply. The value of the context argument will be available in `buffer->requester_data.context` for each buffer. This item is not used by the selection library; it is provided for the convenience of the client. When the reply has been completely processed (or when the consume proc returns something other than `SELN_SUCCESS`), `seln_query` returns.

```

void
seln_report_event(client, event)
    Seln_client_node *client;
    struct inputevent *event;

#define SELN_REPORT(event) seln_report_event(0, event)

```

This is a high-level procedure for informing the selection service of a function key transition which may affect the selection. It incorporates some of the policy of the standard user interface, and provides a more convenient interface to `seln_inform()`.

Client is the client handle returned from `seln_create`; it may be 0 if the client guarantees it will not need to respond to the function transition. Event is a pointer to the `struct inputevent` which reports the transition. `seln_report_event()` generates a corresponding call to `seln_inform()`, and, if the returned struct is not null, passes it to the client's *function_proc* callback procedure (described above under *Callback Procedures*).

`SELN_REPORT` is a macro which takes just an input-event pointer, and calls `seln_report_event` with 0 as a first argument.

```

Seln_result
seln_request(holder, buffer)
    Seln_holder      *holder;
    Seln_request      *buffer;

```

`Seln_request` is the low-level (policy-independent) mechanism for retrieving information about a selection from the process which holds it. Most clients will access it only indirectly, through `seln_ask` or `seln_query`.

`Seln_request` takes a pointer to a holder (as returned by `seln_inquire`), and a request constructed in `*buffer`. The request is transmitted to the indicated selection holder, and the buffer rewritten with its response. Failures in the RPC mechanism will cause a `SELN_FAILED` return; if the process of the addressed holder is no longer active, the return value will be `SELN_NON_EXIST`.

Clients which call `seln_request` directly will find it most convenient to initialize the buffer by a call to `seln_init_request`.

Request attributes which are not recognized by the selection holder will be returned as the value of the attribute `SELN_UNRECOGNIZED`. Responses should be provided in the order requests were encountered.

```

int
seln_same_holder(holder1, holder2)
    Seln_holder  *holder1, *holder2;

```

This predicate returns TRUE if `holder1` and `holder2` refer to the same selection client.

```

int
seln_secondary_exists(buffer)
    Seln_function_buffer  *buffer;

```

This predicate returns TRUE if the function buffer indicates that a secondary selection existed at the time the function key went up.

```

int
seln_secondary_made(buffer)
    Seln_function_buffer  *buffer;

```

This predicate returns TRUE if the function buffer indicates that a secondary selection was made some time since the function key went down (although it may have been cancelled before the key went up).

```

void
seln_use_test_service()

```


The application is set to communicate with a test version of the Selection Service, rather than the standard production version. This call should be made before any selection client is created; this normally means before subwindows in the application process are created.

```
void  
seln_use_timeout(seconds)  
    int    seconds;
```

The default timeout on subsequent RPC calls from this process is changed to be seconds long.

```
void  
seln_yield_all()
```

This procedure inquires the holders of all selection, and for each which is held by a client in the calling process, sends a yield request to that client and a Done to the service. It should be called by applications which are about to exit, or to undertake lengthy computations during which they will be unable to respond to requests concerning selections they hold.

9.9. Common Request Attributes

The following is an annotated listing of
<suntool/selection_attributes.h>.

```

/*  @(#)selection_attributes.h 1.10 85/09/05    */

#ifndef suntool_selection_attributes_DEFINED
#define suntool_selection_attributes_DEFINED

/*
 * Copyright (c) 1985 by Sun Microsystems, Inc.
 */

#include <sunwindow/attr.h>
/*
 * Common requests a client may send to a selection-holder
 */
#define ATTR_PKG_SELECTION  ATTR_PKG_SELN_BASE

#define SELN_ATTR(type, n)  ATTR(ATTR_PKG_SELECTION, type, n)

#define SELN_ATTR_LIST(list_type, type, n)  \
    ATTR(ATTR_PKG_SELECTION, ATTR_LIST_INLINE(list_type, type), n)

```



```

/*
 *      Attributes of selections
 */

typedef enum      {

    /* Simple attributes
     */
    SELN_REQ_BYTESIZE      = SELN_ATTR(ATTR_INT,          1),
    /* value is an int giving the number of bytes in the
     * selection's ascii contents
     */
    SELN_REQ_CONTENTS_ASCII = SELN_ATTR_LIST(ATTR_NULL, ATTR_CHAR, 2),
    /* value is a null-terminated list of 4-byte words containing
     * the selection's ascii contents. The last word containing
     * a character of the selection should be followed by a
     * terminator word whose value is 0. If the last word of
     * contents is not full, it should be padded out with NULs */

    SELN_REQ_CONTENTS_PIECES = SELN_ATTR_LIST(ATTR_NULL, ATTR_CHAR, 3),
    /* value is a null-terminated list of 4-byte words containing
     * the selection's contents described in the textsw's
     * piece-table format.
     */
    SELN_REQ_FIRST          = SELN_ATTR(ATTR_INT,          4),
    /* value is an int giving the number of bytes which precede
     * the first byte of the selection.
     */
    SELN_REQ_FIRST_UNIT     = SELN_ATTR(ATTR_INT,          5),
    /* value is an int giving the number of units of the selection's
     * current level (line, paragraph, etc.) which precede the
     * first unit of the selection.
     */
    SELN_REQ_LAST          = SELN_ATTR(ATTR_INT,          6),
    /* value is an int giving the byte index of the last byte
     * of the selection.
     */
    SELN_REQ_LAST_UNIT      = SELN_ATTR(ATTR_INT,          7),
    /* value is an int giving the unit index of the last unit
     * of the selection at its current level.
     */
    SELN_REQ_LEVEL          = SELN_ATTR(ATTR_INT,          8),
    /* value is an int giving the current level of the selection
     * (See below for #defines of the most useful levels.)
     */
    SELN_REQ_FILE_NAME      = SELN_ATTR_LIST(ATTR_NULL, ATTR_CHAR, 9),
    /* value is a null-terminated list of 4-byte words containing
     * the name of the file which holds the selection (when the
     * Selection Service has been asked to hold a selection).
     * The string is represented exactly like ascii contents.
     */

```

```

/* Simple commands (no parameters)
*/
SELN_REQ_COMMIT_PENDING_DELETE
    = SELN_ATTR(ATTR_NO_VALUE, 65),
/* There is no value. The replier is instructed to delete any
 * secondary selection made in pending delete mode. */
SELN_REQ_DELETE
    = SELN_ATTR(ATTR_NO_VALUE, 66),
/* There is no value. The replier is instructed to delete the
 * selection referred to in this request. */
SELN_REQ_RESTORE
    = SELN_ATTR(ATTR_NO_VALUE, 67),
/* There is no value. The replier is instructed to restore the
 * selection referred to in this request, if it has maintained
 * sufficient information to do so. */

/* Other commands
*/
SELN_REQ_YIELD
    = SELN_ATTR(ATTR_ENUM, 97),
/* The value in the request is not meaningful; in the response,
 * the value is a Seln_result which is the replier's
 * return code. The replier is requested to yield the
 * selection referred to in this request. SELN_SUCCESS,
 * SELN_DIDNT_HAVE, and SELN_WRONG_RANK are legitimate
 * responses (the latter comes from a holder asked to
 * yield the primary selection when it knows a function-key
 * is down). */
SELN_REQ_FAKE_LEVEL
    = SELN_ATTR(ATTR_INT, 98),
/* value is an int giving a level to which the selection
 * should be expanded before processing the remainder of
 * this request. The original level should be maintained
 * on the display, however, and restored as the true level
 * on completion of the request */
SELN_REQ_SET_LEVEL
    = SELN_ATTR(ATTR_INT, 99),
/* value is an int giving a level to which the selection
 * should be set. This request should affect the true level */

/* Service debugging commands
*/
SELN_TRACE_ACQUIRE
    = SELN_ATTR(ATTR_BOOLEAN, 193),
SELN_TRACE_DONE
    = SELN_ATTR(ATTR_BOOLEAN, 194),
SELN_TRACE_HOLD_FILE
    = SELN_ATTR(ATTR_BOOLEAN, 195),
SELN_TRACE_INFORM
    = SELN_ATTR(ATTR_BOOLEAN, 196),
SELN_TRACE_INQUIRE
    = SELN_ATTR(ATTR_BOOLEAN, 197),
SELN_TRACE_YIELD
    = SELN_ATTR(ATTR_BOOLEAN, 198),
SELN_TRACE_STOP
    = SELN_ATTR(ATTR_BOOLEAN, 199),
/* value is a boolean (TRUE / FALSE) indicating whether calls
 * to that procedure in the service should be traced.
 * TRACE_INQUIRE also controls tracing on seln_inquire_all(). */
SELN_TRACE_DUMP
    = SELN_ATTR(ATTR_ENUM, 200),
/* value is a Seln_rank, indicating which selection holder
 * should be dumped; SELN_UNSPECIFIED indicates all holders. */

```



```
/* Close bracket so replier can terminate commands
 * like FAKE_LEVEL which have scope
 */
SELN_REQ_END_REQUEST      = SELN_ATTR(ATTR_NO_VALUE,          253),

/* Error returned for failed or unrecognized requests
 */
SELN_REQ_UNKNOWN          = SELN_ATTR(ATTR_INT,              254),
SELN_REQ_FAILED           = SELN_ATTR(ATTR_INT,              255)

}      Seln_attribute;

/* Meta-levels available for use with SELN_REQ_FAKE/SET_LEVEL.
 *      SELN_LEVEL_LINE is "text line bounded by newline characters,
 *                          including only the terminating newline"
 */
typedef enum {
    SELN_LEVEL_FIRST       = 0x40000001,
    SELN_LEVEL_LINE        = 0x40000101,
    SELN_LEVEL_ALL         = 0x40008001,
    SELN_LEVEL_NEXT        = 0x4000F001,
    SELN_LEVEL_PREVIOUS    = 0x4000F002
}      Seln_level;
#endif
```

9.10. Two Program Examples

There are several programs in the *SunView 1 Programmer's Guide* that do a `seln_ask()` for the primary selection. Here are two sample programs that manipulate the selection in more complex ways.

get_selection Code

The following code is from the program *get_selection*, which is part of the release. This program copies the contents of the desired SunView selection to `stdout`. For more information, consult the `get_selection(1)` man page.

```
#ifndef lint
static char  sccsid[] = "@(#)get_selection.c 10.5 86/05/14";
#endif

/*
 * Copyright (c) 1986 by Sun Microsystems, Inc.
 */

#include <stdio.h>
#include <sys/types.h>
#include <suntool/selection_svc.h>
#include <suntool/selection_attributes.h>

static Seln_result  read_proc();

static int          data_read = 0;

static void          quit();

#ifdef STANDALONE
main(argc, argv)
#else
get_selection_main(argc, argv)
#endif STANDALONE
    int             argc;
    char             **argv;
{
    Seln_client      client;
    Seln_holder      holder;
    Seln_rank         rank = SELN_PRIMARY;
    char              context = 0;
    int               debugging = FALSE;

    while (--argc) {
        /* command-line args control rank of desired selection, */
        /* use of a debugging service, and rpc timeout           */
        argv++;
        switch (**argv) {
            case '1':
                rank = SELN_PRIMARY;
                break;
            case '2':
                rank = SELN_SECONDARY;

```



```

        break;
    case '3':
        rank = SELN_SHELF;
        break;
    case 'D':
        seln_use_test_service();
        break;
    case 't':
    case 'T':
        seln_use_timeout(atoi(++argv));
        --argc;
        break;
    default:
        quit("Usage: get_selection [D] [t seconds] [1 | 2 |3]\n");
}
}
/* find holder of desired selection */
holder = seln_inquire(rank);
if (holder.state == SELN_NONE) {
    quit("Selection non-existent, or selection-service failure\n");
}
/* ask for contents, and let callback proc print them */

(void) seln_query(&holder, read_proc, &context,
                 SELN_REQ_CONTENTS_ASCII, 0, 0);

if (data_read)
    exit(0);
else
    exit(1);
}

static void
quit(str)
    char          *str;
{
    fprintf(stderr, str);
    exit(1);
}

/*
 * Procedure called with each buffer of data returned in response
 * to request transmitted by seln_query.
 */
static Seln_result
read_proc(buffer)
    Seln_request   *buffer;
{
    char          *reply;

    /* on first buffer, we have to skip the request attribute,
     * and then make sure we don't repeat on subsequent buffers
     */

```

```
if (*buffer->requester.context == 0) {
    if (buffer == (Seln_request *) NULL ||
        *((Seln_attribute *) buffer->data) != SELN_REQ_CONTENTS_ASCII) {
        quit("Selection holder error -- unrecognized request\n");
    }
    reply = buffer->data + sizeof (Seln_attribute);
    *buffer->requester.context = 1;
} else {
    reply = buffer->data;
}
fputs(reply, stdout);
fflush(stdout);
data_read = 1;
return SELN_SUCCESS;
}
```


seln_demo

The following program, *seln_demo* gets the selection, but it also sets the selection and responds to appropriate queries about it. It isn't an entirely realistic program, since it doesn't provide selection feedback or use function keys.

It displays a panel with several choices and buttons and a text item. You choose the rank of the selection you wish to set or retrieve first. If you are setting the selection, you may also choose whether you want to literally set the selection or provide the name of a file which contains the selection. Then either type in the selection and push the **Set** button, or just push the **Get** button to retrieve the current selection of the type you chose.

The code has three logical sections: the procedures to create and service the panel, the code to set a selection, and the code to get a selection. The routines to set and get the selection are complicated because they are written to allow arbitrary length selections. Try selecting a 3000 byte piece of text; although you can only see 10 characters of it in the text panel item, the entire selection can be retrieved and/or set.

Large Selections

In order to handle large selections, the selection service breaks them into smaller chunks of about 2000 bytes called buffers. The routines you write must be able to handle a buffer and save enough information so that when they are called again with the next buffer, they can pick up where they left off. *seln_demo* uses the context fields provided in the Selection Service data structures to accomplish this.

```

#ifndef lint
static char sccsid[] = "@(#)seln_demo.c 1.5 88/03/14 Copyr 1986 Sun Micro";
#endif
/*
 * seln_demo.c
 *
 * demonstrate how to use the selection service library
 */

#include <stdio.h>
#include <suntool/sunview.h>
#include <suntool/panel.h>
#include <suntool/seln.h>

static Frame frame;
static Panel panel;

int err = 0;

char *malloc();

/*
 * definitions for the panel
 */

static Panel_item text_item, type_item, source_item, mesg_item;
static Panel_item set_item[3], get_item[3];
static void set_button_proc(), get_button_proc(), change_label_proc();

#define PRIMARY_CHOICE      0      /* get/set the primary selection */
#define SECONDARY_CHOICE    1      /* get/set the secondary selection */
#define SHELF_CHOICE        2      /* get/set the shelf */

#define ITEM_CHOICE         0      /* use the text item literally as the
                                   selection */
#define FROMFILE_CHOICE     1      /* use the text item as the name of a
                                   file which contains the selection */

int selection_type = PRIMARY_CHOICE;
int selection_source = ITEM_CHOICE;

char *text_labels[3][2] = {
    {
        "New primary selection:",
        "File containing new primary selection:"
    },
    {
        "New secondary selection:",
        "File containing new secondary selection:"
    },
    {
        "New shelf:",
        "File containing new shelf:"
    }
}

```



```
    }  
};  
  
char *mesg_labels[3][2] = {  
    {  
        "Type in a selection and hit the Set Selection button",  
        "Type in a filename and hit the Set Selection button"  
    },  
    {  
        "Type in a selection and hit the Set Secondary button",  
        "Type in a filename and hit the Set Secondary button"  
    },  
    {  
        "Type in a selection and hit the Set Shelf button",  
        "Type in a filename and hit the Set Shelf button"  
    }  
};  
  
Seln_rank type_to_rank[3] = { SELN_PRIMARY, SELN_SECONDARY, SELN_SHELF };  
  
/*  
 * definitions for selection service handlers  
 */  
  
static Seln_client s_client;    /* selection client handle */  
  
#define FIRST_BUFFER    0  
#define NOT_FIRST_BUFFER    1  
  
char *selection_bufs[3];    /* contents of each of the three selections;  
                             they are set only when the user hits a set  
                             or a get button */  
  
int func_key_proc();  
Seln_result reply_proc();  
Seln_result read_proc();
```

```

/*****
/* main routine
*****/

main(argc, argv)
int argc;
char **argv;
{
    /* create frame first */

    frame = window_create(NULL, FRAME,
                          FRAME_ARGS,      argc, argv,
                          WIN_ERROR_MSG, "Cannot create frame",
                          FRAME_LABEL, "seln_demo",
                          0);

    /* create selection service client before creating subwindows
       (since the panel package also uses selections) */

    s_client = seln_create(func_key_proc, reply_proc, (char *)0);
    if (s_client == NULL) {
        fprintf(stderr, "seln_demo: seln_create failed!\n");
        exit(1);
    }

    /* now create the panel */

    panel = window_create(frame, PANEL,
                          WIN_ERROR_MSG, "Cannot create panel",
                          0);

    init_panel(panel);

    window_fit_height(panel);

    window_fit_height(frame);

    window_main_loop(frame);

    /* yield any selections we have and terminate connection with the
       selection service */

    seln_destroy(s_client);
}

```



```

/*****
/* routines involving setting a selection */
*****/

/*
 * acquire the selection type specified by the current panel choices;
 * this will enable requests from other clients which want to get
 * the selection's value, which is specified by the source_item and text_item
 */

static void
set_button_proc(/* args ignored */)
{
    Seln_rank ret;
    char *value = (char *)panel_get_value(text_item);

    if (selection_source == FROMFILE_CHOICE) {
        /* set the selection from a file; the selection service will
           actually acquire the selection and handle all requests */

        if (seln_hold_file(type_to_rank[selection_type], value)
            != SELN_SUCCESS) {
            panel_set(mesg_item, PANEL_LABEL_STRING,
                "Could not set selection from named file!", 0);
            err++;
        } else if (err) {
            panel_set(mesg_item, PANEL_LABEL_STRING,
                mesg_labels[selection_type][selection_source], 0);
            err = 0;
        }
        return;
    }
    ret = seln_acquire(s_client, type_to_rank[selection_type]);

    /* check that the selection rank we received is the one we asked for */

    if (ret != type_to_rank[selection_type]) {
        panel_set(mesg_item, PANEL_LABEL_STRING,
            "Could not acquire selection!", 0);
        err++;
        return;
    }

    set_selection_value(selection_type, selection_source, value);
}

/*
 * copy the new value of the appropriate selection into its
 * buffer so that if the user changes the text item and/or the current
 * selection type, the selection won't mysteriously change
 */

set_selection_value(type, source, value)

```

```

int type, source;
char *value;
{
    if (selection_bufs[type] != NULL)
        free(selection_bufs[type]);
    selection_bufs[type] = malloc(strlen(value) + 1);
    if (selection_bufs[type] == NULL) {
        panel_set(mesg_item, PANEL_LABEL_STRING, "Out of memory!", 0);
        err++;
    } else {
        strcpy(selection_bufs[type], value);
        if (err) {
            panel_set(mesg_item, PANEL_LABEL_STRING,
                      mesg_labels[type][source], 0);
            err = 0;
        }
    }
}

/*
 * func_key_proc
 *
 * called by the selection service library whenever a change in the state of
 * the function keys requires an action (for instance, put the primary
 * selection on the shelf if the user hit PUT)
 */

func_key_proc(client_data, args)
char *client_data;
Seln_function_buffer *args;
{
    Seln_holder *holder;

    /* use seln_figure_response to decide what action to take */

    switch (seln_figure_response(args, &holder)) {
    case SELN_IGNORE:
        /* don't do anything */
        break;
    case SELN_REQUEST:
        /* expected to make a request */
        break;
    case SELN_SHELF:
        /* put the primary selection (which we should have) on the
           shelf */
        if (seln_acquire(s_client, SELN_SHELF) != SELN_SHELF) {
            panel_set(mesg_item, PANEL_LABEL_STRING,
                      "Could not acquire shelf!", 0);
            err++;
        } else {
            shelve_primary_selection();
        }
        break;
    }
}

```



```

        case SELN_FIND:
            /* do a search */
            break;
        case SELN_DELETE:
            /* do a delete */
            break;
    }
}

shelve_primary_selection()
{
    char *value = selection_bufs[PRIMARY_CHOICE];

    if (selection_bufs[SHELF_CHOICE] != NULL)
        free(selection_bufs[SHELF_CHOICE]);
    selection_bufs[SHELF_CHOICE] = malloc(strlen(value)+1);
    if (selection_bufs[SHELF_CHOICE] == NULL) {
        panel_set(msg_item, PANEL_LABEL_STRING, "Out of memory!", 0);
        err++;
    } else {
        strcpy(selection_bufs[SHELF_CHOICE], value);
    }
}

/*
 * reply_proc
 *
 * called by the selection service library whenever a request comes from
 * another client for one of the selections we currently hold
 */

Seln_result
reply_proc(item, context, length)
Seln_attribute item;
Seln_replier_data *context;
int length;
{
    int size, needed;
    char *seln, *destp;

    /* determine the rank of the request and choose the
       appropriate selection */

    switch (context->rank) {
    case SELN_PRIMARY:
        seln = selection_bufs[PRIMARY_CHOICE];
        break;
    case SELN_SECONDARY:
        seln = selection_bufs[SECONDARY_CHOICE];
        break;
    case SELN_SHELF:
        seln = selection_bufs[SHELF_CHOICE];
        break;
    }
}

```

```

default:
    seln = NULL;
}

/* process the request */

switch (item) {
case SELN_REQ_CONTENTS_ASCII:
    /* send the selection */

    /* if context->context == NULL then we must start sending
       this selection; if it is not NULL, then the selection
       was too large to fit in one buffer and this call must
       send the next buffer; a pointer to the location to start
       sending from was stored in context->context on the
       previous call */

    if (context->context == NULL) {
        if (seln == NULL)
            return(SELN_DIDNT_HAVE);
        context->context = seln;
    }
    size = strlen(context->context);
    destp = (char *)context->response_pointer;

    /* compute how much space we need: the length of the selection
       (size), plus 4 bytes for the terminating null word, plus 0
       to 3 bytes to pad the end of the selection to a word
       boundary */

    needed = size + 4;
    if (size % 4 != 0)
        needed += 4 - size % 4;
    if (needed <= length) {
        /* the entire selection fits */
        strcpy(destp, context->context);
        destp += size;
        while ((int)destp % 4 != 0) {
            /* pad to a word boundary */
            *destp++ = '\0';
        }
        /* update selection service's pointer so it can
           determine how much data we are sending */
        context->response_pointer = (char **)destp;
        /* terminate with a NULL word */
        *context->response_pointer++ = 0;
        return(SELN_SUCCESS);
    } else {
        /* selection doesn't fit in a single buffer; rest
           will be put in different buffers on subsequent
           calls */
        strncpy(destp, context->context, length);
        destp += length;
    }
}

```



```
        context->response_pointer = (char **)destp;
        context->context += length;
        return(SELN_CONTINUED);
    }
case SELN_REQ_YIELD:
    /* deselect the selection we have (turn off highlight, etc.) */

    *context->response_pointer++ = (char *)SELN_SUCCESS;
    return(SELN_SUCCESS);
case SELN_REQ_BYTESIZE:
    /* send the length of the selection */

    if (seln == NULL)
        return(SELN_DIDNT_HAVE);
    *context->response_pointer++ = (char *)strlen(seln);
    return(SELN_SUCCESS);
case SELN_REQ_END_REQUEST:
    /* all attributes have been taken care of; release any
       internal storage used */
    return(SELN_SUCCESS);
    break;
default:
    /* unrecognized request */
    return(SELN_UNRECOGNIZED);
}
/* NOTREACHED */
}
```

```

/*****
/* routines involving getting a selection */
*****/

/*
 * get the value of the selection type specified by the current panel choices
 * from whichever client is currently holding it
 */

static void
get_button_proc(/* args ignored */)
{
    Seln_holder holder;
    int len;
    char context = FIRST_BUFFER; /* context value used when a very long
                                   message is received; see procedure
                                   comment for read_proc */

    if (err) {
        panel_set(mesg_item, PANEL_LABEL_STRING,
                  mesg_labels[selection_type][selection_source], 0);
        err = 0;
    }

    /* determine who has the selection of the rank we want */

    holder = seln_inquire(type_to_rank[selection_type]);
    if (holder.state == SELN_NONE) {
        panel_set(mesg_item, PANEL_LABEL_STRING,
                  "You must make a selection first!", 0);
        err++;
        return;
    }

    /* ask for the length of the selection and then the actual
       selection; read_proc actually reads it in */

    (void) seln_query(&holder, read_proc, &context,
                     SELN_REQ_BYTESIZE, 0,
                     SELN_REQ_CONTENTS_ASCII, 0,
                     0);

    /* display the selection in the panel */

    len = strlen(selection_bufs[selection_type]);
    if (len > (int)panel_get(text_item, PANEL_VALUE_STORED_LENGTH))
        panel_set(text_item, PANEL_VALUE_STORED_LENGTH, len, 0);
    panel_set_value(text_item, selection_bufs[selection_type]);
}

/*
 * called by seln_query for every buffer of information received; short

```



```

* messages (under about 2000 bytes) will fit into one buffer; for larger
* messages, read_proc will be called with each buffer in turn; the context
* pointer passed to seln_query is modified by read_proc so that we will know
* if this is the first buffer or not
*/

```

```

Seln_result
read_proc(buffer)
Seln_request *buffer;
{
    char *reply;    /* pointer to the data in the buffer received */
    unsigned len;   /* amount of data left in the buffer */
    int bytes_to_copy;
    static int selection_have_bytes; /* number of bytes of the selection
                                     which have been read; cumulative over all calls for
                                     the same selection (it is reset when the first
                                     buffer of a selection is read) */
    static int selection_len;        /* total number of bytes in the current
                                     selection */

    if (*buffer->requester.context == FIRST_BUFFER) {

        /* this is the first buffer */

        if (buffer == (Seln_request *)NULL) {
            panel_set(mesg_item, PANEL_LABEL_STRING,
                      "Error reading selection - NULL buffer", 0);
            err++;
            return(SELN_UNRECOGNIZED);
        }
        reply = buffer->data;
        len = buffer->buf_size;

        /* read in the length of the selection */

        if (*(Seln_attribute *)reply != SELN_REQ_BYTESIZE) {
            panel_set(mesg_item, PANEL_LABEL_STRING,
                      "Error reading selection - unrecognized request",
                      0);
            err++;
            return(SELN_UNRECOGNIZED);
        }
        reply += sizeof(Seln_attribute);
        len = buffer->buf_size - sizeof(Seln_attribute);
        selection_len = *(int *)reply;
        reply += sizeof(int); /* this only works since an int is 4
                               bytes; all values must be padded to
                               4-byte word boundaries */
        len -= sizeof(int);

        /* create a buffer to store the selection */

        if (selection_bufs[selection_type] != NULL)

```

```

        free(selection_bufs[selection_type]);
        selection_bufs[selection_type] = malloc(selection_len + 1);
        if (selection_bufs[selection_type] == NULL) {
            panel_set(msg_item, PANEL_LABEL_STRING,
                      "Out of memory!", 0);
            err++;
            return(SELN_FAILED);
        }
        selection_have_bytes = 0;

        /* start reading the selection */

        if (*(Seln_attribute *)reply != SELN_REQ_CONTENTS_ASCII) {
            panel_set(msg_item, PANEL_LABEL_STRING,
                      "Error reading selection - unrecognized request",
                      0);
            err++;
            return(SELN_UNRECOGNIZED);
        }
        reply += sizeof(Seln_attribute);
        len -= sizeof(Seln_attribute);
        *buffer->requester.context = NOT_FIRST_BUFFER;
    } else {
        /* this is not the first buffer, so the contents of the buffer
           is just more of the selection */

        reply = buffer->data;
        len = buffer->buf_size;
    }

    /* copy data from the received buffer to the selection buffer
       allocated above */

    bytes_to_copy = selection_len - selection_have_bytes;
    if (len < bytes_to_copy)
        bytes_to_copy = len;
    strncpy(&selection_bufs[selection_type][selection_have_bytes],
            reply, bytes_to_copy);
    selection_have_bytes += bytes_to_copy;
    if (selection_have_bytes == selection_len)
        selection_bufs[selection_type][selection_len] = '\0';
    return(SELN_SUCCESS);
}

```



```

/*****
/* panel routines
*****/

/* panel initialization routine */

init_panel(panel)
Panel panel;
{
    msg_item = panel_create_item(panel, PANEL_MESSAGE,
                                PANEL_LABEL_STRING,
                                msg_labels[PRIMARY_CHOICE][ITEM_CHOICE],
                                0);
    type_item = panel_create_item(panel, PANEL_CYCLE,
                                PANEL_LABEL_STRING,    "Set/Get: ",
                                PANEL_CHOICE_STRINGS,  "Primary Selection",
                                                "Secondary Selection",
                                                "Shelf",
                                                0,
                                PANEL_NOTIFY_PROC,    change_label_proc,
                                PANEL_LABEL_X,        ATTR_COL(0),
                                PANEL_LABEL_Y,        ATTR_ROW(1),
                                0);
    source_item = panel_create_item(panel, PANEL_CYCLE,
                                PANEL_LABEL_STRING,    "Text item contains:",
                                PANEL_CHOICE_STRINGS,  "Selection",
                                                "Filename Containing Selection",
                                                0,
                                PANEL_NOTIFY_PROC,    change_label_proc,
                                0);
    text_item = panel_create_item(panel, PANEL_TEXT,
                                PANEL_LABEL_STRING,
                                text_labels[PRIMARY_CHOICE][ITEM_CHOICE],
                                PANEL_VALUE_DISPLAY_LENGTH, 20,
                                0);
    set_item[0] = panel_create_item(panel, PANEL_BUTTON,
                                PANEL_LABEL_IMAGE,    panel_button_image(panel,
                                                "Set Selection", 15,0),
                                PANEL_NOTIFY_PROC,    set_button_proc,
                                PANEL_LABEL_X,        ATTR_COL(0),
                                PANEL_LABEL_Y,        ATTR_ROW(5),
                                0);
    set_item[1] = panel_create_item(panel, PANEL_BUTTON,
                                PANEL_LABEL_IMAGE,    panel_button_image(panel,
                                                "Set Secondary", 15,0),
                                PANEL_NOTIFY_PROC,    set_button_proc,
                                PANEL_LABEL_X,        ATTR_COL(0),
                                PANEL_LABEL_Y,        ATTR_ROW(5),
                                PANEL_SHOW_ITEM,      FALSE,
                                0);
    set_item[2] = panel_create_item(panel, PANEL_BUTTON,
                                PANEL_LABEL_IMAGE,    panel_button_image(panel,
                                                "Set Shelf", 15,0),

```

```

        PANEL_NOTIFY_PROC,      set_button_proc,
        PANEL_LABEL_X,          ATTR_COL(0),
        PANEL_LABEL_Y,          ATTR_ROW(5),
        PANEL_SHOW_ITEM,        FALSE,
        0);
get_item[0] = panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE,      panel_button_image(panel,
                                "Get Selection", 15,0),
        PANEL_NOTIFY_PROC,      get_button_proc,
        PANEL_LABEL_X,          ATTR_COL(20),
        PANEL_LABEL_Y,          ATTR_ROW(5),
        0);
get_item[1] = panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE,      panel_button_image(panel,
                                "Get Secondary", 15,0),
        PANEL_NOTIFY_PROC,      get_button_proc,
        PANEL_SHOW_ITEM,        FALSE,
        PANEL_LABEL_X,          ATTR_COL(20),
        PANEL_LABEL_Y,          ATTR_ROW(5),
        0);
get_item[2] = panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE,      panel_button_image(panel,
                                "Get Shelf", 15,0),
        PANEL_NOTIFY_PROC,      get_button_proc,
        PANEL_SHOW_ITEM,        FALSE,
        PANEL_LABEL_X,          ATTR_COL(20),
        PANEL_LABEL_Y,          ATTR_ROW(5),
        0);
}

/*
 * change the label of the text item to reflect the currently chosen selection
 * type
 */

static void
change_label_proc(item, value, event)
Panel_item item;
int value;
Event *event;
{
    int old_selection_type = selection_type;

    selection_type = (int)panel_get_value(type_item);
    selection_source = (int)panel_get_value(source_item);
    panel_set(text_item, PANEL_LABEL_STRING,
        text_labels[selection_type][selection_source], 0);
    panel_set(msg_item, PANEL_LABEL_STRING,
        msg_labels[selection_type][selection_source], 0);
    if (old_selection_type != selection_type) {
        panel_set(set_item[old_selection_type],
            PANEL_SHOW_ITEM, FALSE, 0);
        panel_set(set_item[selection_type],

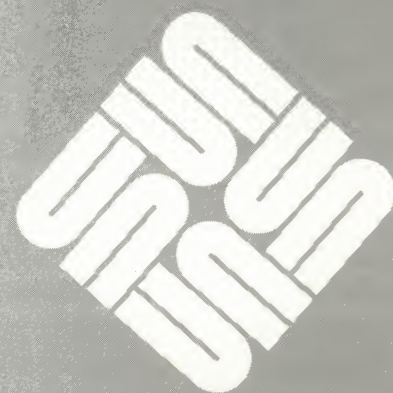
```



```
        PANEL_SHOW_ITEM, TRUE, 0);  
panel_set(get_item[old_selection_type],  
        PANEL_SHOW_ITEM, FALSE, 0);  
panel_set(get_item[selection_type],  
        PANEL_SHOW_ITEM, TRUE, 0);  
    }  
}
```

The User Defaults Database

The User Defaults Database	145
Why a Centralized Database?	145
10.1. Overview	146
Master Database Files	146
Private Database Files	146
10.2. File Format	147
Option Names	147
Option Values	148
Distinguished Names	148
\$Help	148
\$Enumeration	148
\$Message	149
10.3. Creating a .d File: Example	149
10.4. Retrieving Option Values	150
Retrieving String Values	150
Retrieving Integer Values	150
Retrieving Character Values	151
Retrieving Boolean Values	151
Retrieving Enumerated Values	152
Searching for Specific Symbols	152
Searching for Specific Values	153
Retrieving all Values in the Database	153
10.5. Conversion Programs	154



10.6. Property Sheets	155
10.7. Error Handling	156
<i>Error_Action</i>	156
<i>Maximum_Errors</i>	157
<i>Test_Mode</i>	157
10.8. Interface Summary	157
10.9. Example Program: <i>filer</i> Defaults Version	160

The User Defaults Database

Many UNIX programs are *customizable* in that the user can modify their behavior by setting certain parameters checked by the program at startup time. This approach has been extended in SunView to include facilities used by many applications, such as menus, text and scrollbars, as well as applications.

The SunView *user defaults database* is a centralized database for maintaining customization information about different programs and facilities.

This chapter is addressed to programmers who want their programs to make use of the defaults database. For a discussion of the user interface to the defaults database, read the `defaultsedit(1)` manual page in the *SunOS Reference Manual* and see the section on `defaultsedit` in the *SunView 1 Beginner's Guide*.

In this chapter, customizable parameters are referred to as *options*; the values they can be set to are referred to as *values*.

All definitions necessary to use the defaults database may be obtained by including the file `<sunwindow/defaults.h>`.

Why a Centralized Database?

Traditionally, each customizable program has a corresponding *customization file* in the user's home directory. The program reads its customization file at startup time to get the values the user has specified.

Examples of customizable programs are `mail(1)`, `csh(1)`, and `sunview(1)`. The corresponding customization files are `.mailrc`, `.cshrc`, and `.sunview`.

While this method of handling customization works well enough, it can become confusing to the user because:

- Since the information is scattered among programs, it's difficult for the user to determine what options she or he can set.
- Since the format of each customization file is different, the user must find and read documentation for each program he or she wants to customize.
- Even after the user has located the customization file and become familiar with its format, it's often difficult for the user to determine what the legal values are for a particular option.

SunView addresses these problems by providing a centralized database which can be used by any customizable program. The user can view and modify the options in the defaults database with the interactive program `defaultsedit(1)`.

10.1. Overview

The defaults database actually consists of a single *master database* and a *private database* for each user.

The master database contains all the options for each program which uses the defaults database. For each option, the *default value* is given.

The user's private database contains the values she or he has specified using `defaultsedit`. An option's value in the private database takes precedence over the option's default value in the master database.

Application programs retrieve values from the database using the routines described later in this chapter. These routines first search the user's private database for the value. If the value is not found in the private database, then the default value from the master database is returned. Each of these routines specify a fall-back default value which is used if neither database contains the value. It should match the value in the master database.

Master Database Files

The master database is stored in the directory `/usr/lib/defaults` as a number of individual files, each containing the options for one program or package. These files are created with a text editor by the author of the program or package; see Section 10.3, *Creating a .d File: Example*, later in this chapter. By convention, the file name is the capitalized name of the program or package, with the suffix `.d` — `Mail.d`, `SunView.d`, `Menu.d`, etc.

The defaults database itself has two options you can set using `defaultsedit` to control where the master database resides:

- *Directory* is provided so that a group may have its own master database directory in which to do development independently of the standard `/usr/lib/defaults` directory.
- *Private_Directory* is provided so that an individual developer may have his own private master database for development. Note that this directory must have copies (or symbolic links) to all of the `.d` files in `/usr/lib/defaults`, or accesses to the absent files will result in run-time errors.

When the master database is accessed, the defaults routines look for the appropriate `.d` file first in the *Private_directory* (if specified). If the file is not found or the directory not specified, then if a *Directory* is specified it is searched, otherwise the default directory, `/usr/lib/defaults`, is searched.

Private Database Files

A user's private database is stored in the file `.defaults` in the user's home directory. This is where changes the user makes using `defaultsedit` are recorded.²³

²³ There is rarely any need for the user to edit his `.defaults` file by hand — it is automatically created

There is an option called *Private_only* which allows the user to disable the reading of the master database entirely, thereby reducing program startup time. Note that for this to work, you must make sure that the fall-back values you specify in your program exactly match the values in the master database.

10.2. File Format

The format for both master and private database files is identical.

The first line in the file contains a version number.²⁴ The rest of the file consists of a number of lines, each line contains either an option name with its associated value or a comment, preceded by a semi-colon (;). Blank lines are also legal.

Option Names

The option names are organized hierarchically, just like files in a file system. Names must always start with a slash character, (/), and each level in the naming hierarchy is separated from the previous level by a slash character. Each name consists of one or more letters (A-Z, a-z), digits (0-9), dollar signs (\$), and underscores (_). By convention, the first letter of each name is capitalized.²⁵

There are two shorthand notations for option names. First, whenever a line does not start with a slash, the previous node is prepended to the name (this is similar to the treatment of path names in UNIX). So

```
/SunView/Font
    $Help
```

is equivalent to

```
/SunView/Font
/SunView/Font/$Help
```

The second shorthand convention is that any time two slashes in a row are encountered, the option name previously defined at that level is assumed. Each pair of slashes corresponds to one name. So

```
/SunView/Font
//Walking_Menus
//Icon_gravity
```

is equivalent to

```
/SunView/Font
/SunView/Walking_Menus
    /SunView/Icon_gravity
```

and updated by `defaultsedit`. The one time the user needs to edit his `.defaults` file by hand is to disable the defaults *Testmode* option once it has been enabled. See the discussion in Section 10.7, *Error Handling*, below.

²⁴ The version number is included so that if any incompatible changes are made to the default database format in the future, the library routines can tell when they encounter an older file format.

²⁵ This convention is just for readability — internally all names are converted to lower case, so the defaults database is insensitive to case.

and

```
/SunView/Font/Bold
///Italic
///Size
```

is equivalent to

```
/SunView/Font/Bold
/SunView/Font/Italic
/SunView/Font/Size
```

Option Values

All option values are stored as strings. They have the same syntax as quoted strings in C. In particular, the backslash character (\) is used as an escape character for inserting other characters into the quoted string. The following backslash escapes are recognized:

Table 10-1 *Defaults Metacharacters*

<i>Backslash Escape</i>	<i>Prints as:</i>
\\	Backslash
\"	Double quote
\'	Single quote
\n	Newline
\t	Tab
\b	Backspace
\r	Carriage return
\f	Form feed
\ddd	3 digit octal number specifying a single character

Option values can be up to 10,000 characters long.

Distinguished Names

There are several distinguished names used by `defaultsedit`. See the next section for an example illustrating their usage.

\$Help

\$Help allows you to add an explanatory string to be displayed by `defaultsedit` for each option.

\$Enumeration

An *enumerated option* is one in which the values are explicitly given, such as {True, False}, {Yes, No}, {North, South, East, West} etc.²⁶ The user selects one of the values using `defaultsedit`.

The way that `defaultsedit` knows that it has encountered an enumerated option is by the level name \$Enumeration. The values for the enumerated option follow at the same level. Note that you can specify a help string for the entire enumerated option, as well as specifying the value.

²⁶ There is no limit to the number of values an enumerated option can have.

\$Message

\$Message allows you to add a one-line message to be displayed by `defaultsedit`. Use this to make more readable the display of a category with many options by setting off related options with blank lines or headings.

10.3. Creating a .d File: Example

Adding options for a new program to the database corresponds to adding a new first-level option name in the master database, and appears to the user as a new *category* in `defaultsedit`. You do this by creating the appropriate .d file in `/usr/lib/defaults`. If the file is in the correct format, and ends in .d, then `defaultsedit` will automatically display it as a new category.

Let's create such a file for a game called "Space Wars." The options are: the number of friendly and enemy ships, whether or not stars attract ships, the name of the user's ship, and the direction that ships enter the window from.

To conform to the naming convention for master database files, we add the suffix .d to the first-level option name, yielding the filename `SpaceWar.d`:

```
SunDefaults_Version 2
/SpaceWar
    $Help          "A space ship battle game"
//Friends          "15"
    $Help          "Number of friendly ships"
//Enemies          "15"
    $Help          "Number of enemy ships"
//Gravity          "Yes"
    $Help          "Affects whether star attract ships"
    $Enumeration   ""
    Yes            ""
    Yes/$Help      "Stars attract ships"
    No             ""
    No/$Help       "Ships are immune to attraction"
//Name             "Battlestar"
    $Help          "Name of your space ship"
//Direction        "North"
    $Help          "Starting window border"
    $Enumeration   ""
    North          ""
    North/$Help    "Ships start at north window border"
    South          ""
    South/$Help    "Ships start at south window border"
    East           ""
    East/$Help     "Ships start at east window border"
    West           ""
    West/$Help     "Ships start at west window border"
```

Note that the highest-level option name, `/SpaceWar`, has no associated value, since it wouldn't make sense to have one. If a database routine tries to access an option which has no value, the special string `DEFAULTS_UNDEFINED` will be returned.

10.4. Retrieving Option Values

A simple programmatic interface is provided to retrieve option values from the defaults database. All values are stored as strings, and may be retrieved with `defaults_get_string()`. For convenience, similar get routines are provided to retrieve values as integers, characters, or enumerated types. The get routines are described below.

Retrieving String Values

To retrieve a string value, use:

```
char *
defaults_get_string(option_name, default_value, 0)
    char *option_name;
    char *default_value;
```

`option_name` is the name of the option whose value will be retrieved. `default` is a value to return if the option is not found in the database or if the database itself cannot be accessed for any reason. Note that this value should match the default value in the master database. The final argument to all `defaults_get*()` routines is zero.²⁷ In our Space Wars example in the previous section, we would call

```
ship = defaults_get_string("/SpaceWar/Name",
                          "Battlestar", 0);
```

On return, `ship` would point to the string `Battlestar`.

Suppose you misspelled the option name `Name` as `Nane`. Since `/SpaceWar/Nane` is not in the defaults database, the fall-back value of `Battlestar` will be returned and an error message may be output.²⁸

Retrieving Integer Values

To retrieve an integer value, use:

```
int
defaults_get_integer(option_name, default_value, 0)
    char *option_name;
    int default_value;
```

This function gets the option value associated with `option_name`, treats it as a decimal integer, and returns the integer value. For example, the string `"17"` parses into the number 17 and the string `"-123"` parses into the number -123. If `option_name` can't be found, or its associated value can't be parsed, the integer passed in for `default_value` is returned. For example, the call

```
defaults_get_integer("/SpaceWar/Enemies", 15, 0);
```

will return the integer 15, since `"Enemies"` was misspelled as `"Enemys"`.

²⁷ This third argument is not currently used. It is necessary for compatibility with future releases of the defaults database package, which may use the third argument to return status information.

²⁸ Whether or not the database retrieval routines generate error messages on error conditions depends on the setting of the option `Error_Action`. See Section 10.7, *Error Handling*, below.

The function `defaults_get_integer_check()` is the same as `defaults_get_integer()`, except that it checks that the returned value is within a specified range:

```
int
defaults_get_integer_check(option_name, default_value,
                           min, max, 0)
char *option_name;
int   default_value;
int   min, max;
```

If the option value is not between min and max, the integer passed in for `default_value` is returned and an error message may be output.

Retrieving Character Values

To retrieve a character value, use:

```
int
defaults_get_character(option_name, default_value, 0)
char *option_name;
char  default_value;
```

`defaults_get_character()` returns the first character from the option value. If the option value contains more than one character, the character passed in for `default_value` is returned and an error message is output.

Retrieving Boolean Values

To retrieve a boolean value,²⁹ use:

```
Bool
defaults_get_boolean(option_name, default_value, 0)
char *option_name;
Bool  default_value;
```

`defaults_get_boolean()` returns True if the option value is True, Yes, Enabled, Set, Activated, or 1 and False if the option value is False, No, Off, Disabled, Reset, Cleared, Deactivated, or 0. If the option value is not one of the above, the value passed in for `default_value` is returned and an error message is output.

²⁹ The definition for Bool, found in <sunwindow/sun.h>, is: `typedef enum {False = 0, True = 1} Bool;`

Retrieving Enumerated Values

You can retrieve enumerated option values with `defaults_get_string()`, then use `strcmp(3)` to test which value was returned. As an alternative, you may find it more convenient to define an enumerated type corresponding to the option values, and use `defaults_get_enum()` to return the option value as the corresponding enum. The definition is:

```
int
defaults_get_enum(option_name, pairs)
    char      *option_name;
    Defaults_pairs pairs[];
```

`pairs` is a pointer to an array of `Defaults_pairs` which contains name-value pairs. `Defaults_pairs` is defined as:

```
typedef struct {
    char      *name;
    int       value;
} Defaults_pairs;
```

The array passed in as `pairs` must be null-terminated.

`defaults_get_enum()` returns the name associated with the value which is the current value of the option. If no match is found, the value associated with the last (null) entry is returned.

The following example, using the direction option for our Space Wars example, illustrates the usage of `defaults_get_enum()`:

```
typedef enum {North, South, East, West} directions;
directions dir;
Defaults_pairs direction_pairs [] = {
    "North", (int) North,
    "South", (int) South,
    "East",  (int) East,
    "West",  (int) West,
    NULL,    (int) North};    /* Error value */

dir = defaults_get_enum("/SpaceWar/Direction", direction_pairs);
```

Searching for Specific Symbols

To probe the defaults database to see whether or not a particular symbol is stored in it, use the `defaults_exists()` routine. This routine will return `TRUE`, if `path_name` has a value defined in the defaults database. Otherwise a value of `FALSE` will be returned.

```
Bool flag1 = defaults_exists("/SpaceWar/Ship_name", NULL);
Bool flag2 = defaults_exists("/SpaceWar/Fred", NULL);
```

`flag1` has a value of `TRUE`; `flag2` has a value of `FALSE`.

Searching for Specific Values

To find the original value of a particular database entry before the client's personalized database overwrites it, use

```
defaults_get_default(path_name, default_value, status)
```

For example, assume that the master database has the entry

```
/SpaceWar/ShipName "Lollipop"
```

and that the client's private database is

```
/SpaceWar/ShipName "Death Avenger"
```

If you call `defaults_get_string("/SpaceWar/ShipName", "<err>", NULL)` it will return `Lollipop`. If, however, the `path_name` is not in the database, then the default value will be returned.

Retrieving all Values in the Database

To search the database to find all of the values in the database, use the routines `defaults_get_child()` and `defaults_get_sibling()`. The routine `defaults_get_child(path_name, status)` returns the simple name of the database entry immediately under `path_name`. If you use `defaults_get_child("/SpaceWar", NULL)` it will return `ShipName`. You can use `sprintf(3S)` to construct the full path name:

```
char temp[1000], *child;
child = defaults_get_child("/SpaceWar", NULL);
if (child == NULL) {
    (void)fprintf(stderr, "Error");
    exit(1);
}
sprintf(temp, "%s/%s", "/SpaceWar", child);
```

`temp` would contain `/SpaceWar/ShipName`. A `NULL` value is returned in there is no child.

`defaults_get_sibling(path_name, status)` returns the simple name of the next database entry immediately following `pathname` at the same level. So, if you use

`defaults_get_sibling("/SpaceWar/ShipName", NULL)` it will return `Framus`. This can be assembled into a full path name using `sprintf(3S)`.

```
char temp[1000], *sibling;
sibling = defaults_get_sibling("/SpaceWar/ShipName");
if (sibling == NULL) {
    (void)fprintf(stderr, "Error");
    exit(1);
}
sprintf(temp, "%s/%s", "/SpaceWar", sibling);
```


The following program will dump the entire contents of the defaults database along with their associated values.

```
void
dump_defaults(path_name, indent)
    char    *path_name;
    char    *indent;
{
    char    temp[1000];
    char    *child;

    (void)printf("%s%s %s0, indent, path_name,
                (defaults_get_string(path_name, "<err>", NULL ));
    child = defaults_get_child(path_name, NULL);
    if (child == NULL) {
        return;
    }
    len = strlen(indent);
    indent[len] = ' ';
    indent[len+1] = ' ';
    (void)sprintf(temp, "%s/%s", path_name, child);
    dump_defaults(temp, indent);
    while (sibling = defaults_get_sibling(temp, NULL)
          != NULL) {
        (void)sprintf(temp, "%s/%s", path_name,
                      sibling);
        (dump_defaults(temp, indent);
    }
}

main()
{
    char indent_buf[100] = "";
    dump_defaults("/", indent);
}
```

10.5. Conversion Programs

The defaults package provides a mechanism to convert from an existing customization file, such as `.mailrc`, to the `.d` format used by `defaultsedit`.

You must write a separate program to do the conversion each way. Specify the name of the program converting from the existing customization file to the defaults format as the value of the `$Specialformat_to_defaults` option in the corresponding `.d` file. The program to go the other way is specified as `$Defaults_to_specialformat`.

As an example, at Sun we have written programs to convert from the traditional `.mailrc` file to the defaults format. The file `/usr/lib/defaults/Mail.d` contains the lines:

```
/Mail                ""
//$Specialformat_to_defaults "mailrc_to_defaults"
//$Defaults_to_specialformat "defaults_to_mailrc"
```

If a program is specified as the value for `$Specialformat_to_defaults`, then `defaultsedit` runs the program the first time it needs to display the options for that category. When the user saves the changes she or he has made to the database, and any changes that were made to the category, then the `$Defaults_to_specialformat` program is run.

To write your own conversion programs, use the following guidelines. Read the customization file into the program. Then, to go from the customization file to `.defaults`, you simply figure out the appropriate option value to set, and set it with the routine `defaults_set_string()`.³⁰ To go the other way, retrieve the value from the defaults database with the appropriate get routine, then make the appropriate change to the customization file.

Note: Conversion programs should use the master database, regardless of the setting of the `defaultsedit` option *Private-only*. To do this, call the function `defaults_special_mode()` as the first statement of your program.

10.6. Property Sheets

Many window programs have property sheets that the end-user can use to modify the behavior of their programs. You may use the defaults database to store the information set by the user.

The following discussion uses code fragments taken from the *default_filer* program printed at the end of this chapter.

The *filer* program has a property sheet that consists of one item, the flags to pass through to the `ls` command. This property is represented as a string. When the property sheet is popped up it is necessary to read the value from the database:

```
char *filer_flags;
filer_flags = defaults_get_string("/Filer/LsFlags", "-l",
                                NULL);
```

When, the user changes the flags property, it is necessary to store the new value into the database. This is done using `defaults_set_string(path_name, new_value, status)`.

For example:

```
filer_flags = code from example;
defaults_set_string("/Filer/LsFlags", filer_flags, NULL);
```

³⁰ `defaults_set_string()` is documented in `<defaults/defaults.h>`.

This code writes the new value into the defaults database in memory. The new value will not be stored in the file system until the user pushes the **Done** button. When this occurs, the routine `defaults_write_changed(file_name, status)` is called. This routine will write any database values in memory that are different from the ones in the master database to the defaults file `file_name`. If `file_name` has a value of `NULL`, then it will be written out to the user's private defaults database file. So,

```
defaults_write_changed(NULL, NULL);
```

will cause the default values to be stored to `~/ .defaults`.

If the user pushes the **Reset** button, then you want to reset the property sheet to be the value that the property sheet had when it first came up. A call to `defaults_reread(path_name, status)` will restore the database under `path_name` from the file system. So,

```
defaults_reread("/Filer", NULL);
```

will restore the defaults database for the filer property sheet. You can obtain the original value by reentering the property sheet display code to obtain the original values.

10.7. Error Handling

The defaults routines report errors by printing messages on the standard error stream `stderr`. The most common cause for getting error messages is that a program that uses the defaults database is copied from somewhere without also copying the associated master defaults database file. While these messages are annoying, in general the program will continue to work, since every routine that accesses the defaults database has a `default_value` argument that will be returned if an option is not present in the database.³¹

Using `defaultsedit`, the user can set two options for the defaults database itself to control error reporting:

Error_Action

Error_Action controls what happens when an error is encountered. Possible values are:

- *Continue*: print an error message and continue execution.
- *Suppress*: no action is taken.
- *Abort*: print an error message and terminate execution on encountering the first error.

Most users will want to set *Error_Action* to either *Continue* or *Suppress*. Use *Suppress* if you are getting all sorts of extraneous defaults error messages. *Abort* is useful for forcing programmers to track down extraneous error messages prior to releasing software to a larger community.

³¹ These error messages are not printed when *Private_only* is *True*.

Maximum_Errors

Maximum_Errors puts a limit on the number of error messages which will be printed regardless of the setting of *Error_Action*.

Test_Mode

The option *Test_Mode* is provided to facilitate the testing of software prior to release to a larger community. Use it to check for incorrect values for the *default_value* argument to the get routines. When *Test_Mode* is set to *Enable*, the defaults database is made inaccessible. In this mode, every time an option value is accessed, a diagnostic message is generated and the value passed in as *default_value* is returned.

Note that once enabled, *Test_Mode* can not be disabled using `defaultsedit`. This is one time when you must edit your `.defaults` file by hand, to set the *Test_Mode* option to *Disabled* (or remove the entry altogether).

10.8. Interface Summary

The following table lists and explains all of the procedures that may be used to make use of the defaults database.

Table 10-2 *Default Procedures*

<i>Routine</i>	<i>Description</i>
Bool <code>defaults_exists(path_name, status)</code> char *path_name; int *status;	Returns TRUE if path_name exists in the database.
Bool <code>defaults_get_boolean(option_name, default, 0)</code> char *option_name; Bool default;	Looks up path_name in the database and returns TRUE if the value is True, Yes, On, Enabled, Set, Activated, or 1. FALSE is returned if the value is False, No, Off, Disabled, Reset, Cleared, Deactivated, or 0. If the value is not one of the above values, then a warning message is displayed and the default is returned.
char <code>defaults_get_character(option_name, default, 0)</code> char *option_name; char default;	Looks up path_name in the defaults database and returns the resulting character value. The default value is returned in an error occurs.
char * <code>defaults_get_child(path_name, status)</code> char *path_name; int *status;	Returns a pointer to the simple name associated with the next sibling of path_name. NULL will be returned if path_name does not exist or if path_name does not have a next sibling

Table 10-2 *Default Procedures—Continued*

<i>Routine</i>	<i>Description</i>
<pre>char * defaults_get_default(path_name, default, status) char *path_name; char *default_value; int *status;</pre>	Returns the value associated with <code>path_name</code> prior to being overridden by the clients private database. <code>default</code> is returned in any error occurs.
<pre>int defaults_get_enum(option_name, pairs, 0) char *option_name; Defaults_pairs *pairs;</pre>	Looks up the values associated with <code>path_name</code> , scans the pairs table, and returns the associated value. If no match can be found, then an error will be generated and the value associated with the last entry will be returned. (See <code>defaults_lookup()</code> .)
<pre>int defaults_get_integer(option_name, default, 0) char *option_name; int default;</pre>	Looks up <code>path_name</code> in the defaults database and returns the resulting integer. The default value is returned if any error occurs.
<pre>int defaults_get_integer_check(option_name, default_value, min, max, 0) char *option_name; int default_value; int min, max;</pre>	Looks up <code>path_name</code> in the defaults database and returns the resulting value. If the value in the database is not between minimum and maximum (inclusive), then an error message will be printed. The default will be returned if an error occurs.
<pre>char * defaults_get_sibling(path_name, status) char *path_name; int *status;</pre>	Returns a pointer to the simple name associated with the next sibling of <code>path_name</code> . <code>NULL</code> will be returned, if <code>path_name</code> does not exist or if <code>path_name</code> does not have an next sibling.
<pre>char * defaults_get_string(option_name, default, 0) char *option_name; char *default;</pre>	Looks up <code>path_name</code> in the defaults database and returns the string value associated with it. The default will be returned if any error occurs.
<pre>void defaults_reread(path_name, status) char *path_name; int *status;</pre>	Rereads the portion of the database associated with <code>path_name</code> .
<pre>defaults_set_character(path_name, value, status) char *path_name; char value; int *status;</pre>	Sets <code>path_name</code> to <code>value</code> , while <code>value</code> is a character.

Table 10-2 *Default Procedures—Continued*

<i>Routine</i>	<i>Description</i>
<pre>void defaults_set_enumeration(path_name, value, status) char *path_name; char *value; int *status;</pre>	Sets path_name to value, where value is a pointer to a string.
<pre>void defaults_set_integer(path_name, value, status) char *path_name; int value; int *status;</pre>	Sets path_name to value, where value is an integer.
<pre>void defaults_set_string(path_name, value, status) char *path_name; char *value; int *status;</pre>	Sets path_name to value where value is a pointer to a string
<pre>void defaults_special_mode()</pre>	Causes the database to behave as if the entire master database has been read into memory prior to reading in the private database. This is done to insure that the order of nodes that defaultsedit presents is the same as that in the .d files, regardless of what the user happens to have set in his or her private database.
<pre>void defaults_write_all(path_name, file_name, status) char *path_name; char *file_name; int *status;</pre>	Writes out all of the data base nodes from path_name and below into file_name. Out_file is the string name of the file to create. If file_name is NULL, then env var DEFAULTS_FILE will be used.
<pre>void defaults_write_changed(file_name, status) char *file_name; int *status;</pre>	Writes out all of the private database entries to file_name. Any time a database node is set, it becomes part of the private database. If the value of the File_Name is NULL, then DEFAULTS_FILE will be used.
<pre>void defaults_write_differences(file_name, status) char *file_name; int *status;</pre>	Writes out all of the database entries that differ from the master database. Out_File is the string name of the file to create. If file_name is NULL, env var DEFAULTS_FILE is used.

10.9. Example Program: filer Defaults Version

The following program is a variation of the *filer* program discussed in the *SunView 1 Programmer's Guide*. It uses some of the defaults procedures discussed in this chapter.

```

/*****
/*                                4.0      filer_default.c                                */
*****/

#include <suntool/sunview.h>
#include <sunwindow/defaults.h>
#include <suntool/panel.h>
#include <suntool/tty.h>
#include <suntool/textsw.h>
#include <suntool/seln.h>
#include <suntool/alert.h>
#include <sys/stat.h>      /* stat call needed to verify existence of files */

/* these objects are global so their attributes can be modified or retrieved */
Frame      base_frame, edit_frame, ls_flags_frame;
Panel      panel, ls_flags_panel;
Tty        ttysw;
Textsw     editsw;
Panel_item  dir_item, fname_item, filing_mode_item, reset_item, done_item;
int         quit_confirmed_from_panel;
char *compose_ls_options();

#define      MAX_FILENAME_LEN      256
#define      MAX_PATH_LEN          1024

char *getwd();

main(argc, argv)
    int     argc;
    char **argv;
{
    static Notify_value filer_destroy_func();
    void      ls_flags_proc();

    base_frame = window_create(NULL, FRAME,
                                FRAME_ARGS,      argc, argv,
                                FRAME_LABEL,      "filer",
                                FRAME_PROPS_ACTION_PROC, ls_flags_proc,
                                FRAME_PROPS_ACTIVE, TRUE,
                                FRAME_NO_CONFIRM, TRUE,
                                0);
    (void) notify_interpose_destroy_func(base_frame, filer_destroy_func);

    create_panel_subwindow();
    create_tty_subwindow();
    create_edit_popup();
    create_ls_flags_popup();
    quit_confirmed_from_panel = 0;

    window_main_loop(base_frame);
    exit(0);
}

```

```

create_tty_subwindow()
{
    ttysw = window_create(base_frame, TTY, 0);
}

create_edit_popup()
{
    edit_frame = window_create(base_frame, FRAME,
                               FRAME_SHOW_LABEL, TRUE,
                               0);
    editsw = window_create(edit_frame, TEXTSW, 0);
}

create_panel_subwindow()
{
    void ls_proc(), ls_flags_proc(), quit_proc(), edit_proc(),
        edit_sel_proc(), del_proc();

    char current_dir[MAX_PATH_LEN];

    panel = window_create(base_frame, PANEL, 0);

    (void) panel_create_item(panel, PANEL_BUTTON,
                             PANEL_LABEL_X,          ATTR_COL(0),
                             PANEL_LABEL_Y,          ATTR_ROW(0),
                             PANEL_LABEL_IMAGE,       panel_button_image(panel, "List Directory", 0, 0),
                             PANEL_NOTIFY_PROC,       ls_proc,
                             0);

    (void) panel_create_item(panel, PANEL_BUTTON,
                             PANEL_LABEL_IMAGE,       panel_button_image(panel, "Set ls flags", 0, 0),
                             PANEL_NOTIFY_PROC,       ls_flags_proc,
                             0);

    (void) panel_create_item(panel, PANEL_BUTTON,
                             PANEL_LABEL_IMAGE,       panel_button_image(panel, "Edit", 0, 0),
                             PANEL_NOTIFY_PROC,       edit_proc,
                             0);

    (void) panel_create_item(panel, PANEL_BUTTON,
                             PANEL_LABEL_IMAGE,       panel_button_image(panel, "Delete", 0, 0),
                             PANEL_NOTIFY_PROC,       del_proc,
                             0);

    (void) panel_create_item(panel, PANEL_BUTTON,
                             PANEL_LABEL_IMAGE,       panel_button_image(panel, "Quit", 0, 0),
                             PANEL_NOTIFY_PROC,       quit_proc,
                             0);

    filing_mode_item = panel_create_item(panel, PANEL_CYCLE,
                                         PANEL_LABEL_X,          ATTR_COL(0),
                                         PANEL_LABEL_Y,          ATTR_ROW(1),
                                         PANEL_LABEL_STRING,      "Filing Mode:",
                                         PANEL_CHOICE_STRINGS,     "Use \"File:\" item",
                                         "Use Current Selection", 0,
                                         0);

    (void) panel_create_item(panel, PANEL_MESSAGE,

```



```

        PANEL_LABEL_X,          ATTR_COL(0),
        PANEL_LABEL_Y,          ATTR_ROW(2),
        0);

    dir_item = panel_create_item(panel, PANEL_TEXT,
        PANEL_LABEL_X,          ATTR_COL(0),
        PANEL_LABEL_Y,          ATTR_ROW(3),
        PANEL_VALUE_DISPLAY_LENGTH, 60,
        PANEL_VALUE,            getwd(current_dir),
        PANEL_LABEL_STRING,      "Directory: ",
        0);

    fname_item = panel_create_item(panel, PANEL_TEXT,
        PANEL_LABEL_X,          ATTR_COL(0),
        PANEL_LABEL_Y,          ATTR_ROW(4),
        PANEL_LABEL_DISPLAY_LENGTH, 60,
        PANEL_LABEL_STRING,      "File: ",
        0);

    window_fit_height(panel);

    window_set(panel, PANEL_CARET_ITEM, fname_item, 0);
}

typedef struct Filer {
    char    *flags;
    char    *path;
};

struct Filer filer_options = {" ", "/Filer/Options"};
struct Filer filer_format = {" 1 ", "/Filer/Format"};
struct Filer filer_sort_order = {" r ", "/Filer/Sort_Order"};
struct Filer filer_sort_criterion = {" tu", "/Filer/Sort_Criterion"};
struct Filer filer_directories = {" d ", "/Filer/Sort_Directories"};
struct Filer filer_recursive = {" R ", "/Filer/Recursive"};
struct Filer filer_file_type = {" F ", "/Filer/File_Type"};
struct Filer filer_dot_files = {" a ", "/Filer/Dot_Files"};

create_ls_flags_popup()
{
    void done_proc();
    void reset_proc();

    ls_flags_frame = window_create(base_frame, FRAME, 0);

    ls_flags_panel = window_create(ls_flags_frame, PANEL, 0);

    panel_create_item(ls_flags_panel, PANEL_MESSAGE,
        PANEL_ITEM_X,          ATTR_COL(14),
        PANEL_ITEM_Y,          ATTR_ROW(0),
        PANEL_LABEL_STRING,     "Options for ls command",
        PANEL_CLIENT_DATA,      &filer_options,
        0);

    panel_create_item(ls_flags_panel, PANEL_CYCLE,
        PANEL_ITEM_X,          ATTR_COL(0),
        PANEL_ITEM_Y,          ATTR_ROW(1),
        PANEL_DISPLAY_LEVEL,    PANEL_CURRENT,

```

```

        PANEL_LABEL_STRING,      "Format:                                ",
        PANEL_CHOICE_STRINGS,    "Short", "Long", 0,
        PANEL_CLIENT_DATA,       &filer_format,
        0);

panel_create_item(ls_flags_panel, PANEL_CYCLE,
        PANEL_ITEM_X,           ATTR_COL(0),
        PANEL_ITEM_Y,           ATTR_ROW(2),
        PANEL_DISPLAY_LEVEL,    PANEL_CURRENT,
        PANEL_LABEL_STRING,      "Sort Order:                                ",
        PANEL_CHOICE_STRINGS,    "Descending", "Ascending", 0,
        PANEL_CLIENT_DATA,       &filer_sort_order,
        0);

panel_create_item(ls_flags_panel, PANEL_CYCLE,
        PANEL_ITEM_X,           ATTR_COL(0),
        PANEL_ITEM_Y,           ATTR_ROW(3),
        PANEL_DISPLAY_LEVEL,    PANEL_CURRENT,
        PANEL_LABEL_STRING,      "Sort criterion:                            ",
        PANEL_CHOICE_STRINGS,    "Name", "Modification Time",
                                   "Access Time", 0,
        PANEL_CLIENT_DATA,       &filer_sort_criterion,
        0);

panel_create_item(ls_flags_panel, PANEL_CYCLE,
        PANEL_ITEM_X,           ATTR_COL(0),
        PANEL_ITEM_Y,           ATTR_ROW(4),
        PANEL_DISPLAY_LEVEL,    PANEL_CURRENT,
        PANEL_LABEL_STRING,      "For directories, list:                    ",
        PANEL_CHOICE_STRINGS,    "Contents", "Name Only", 0,
        PANEL_CLIENT_DATA,       &filer_directories,
        0);

panel_create_item(ls_flags_panel, PANEL_CYCLE,
        PANEL_ITEM_X,           ATTR_COL(0),
        PANEL_ITEM_Y,           ATTR_ROW(5),
        PANEL_DISPLAY_LEVEL,    PANEL_CURRENT,
        PANEL_LABEL_STRING,      "Recursively list subdirectories? ",
        PANEL_CHOICE_STRINGS,    "No", "Yes", 0,
        PANEL_CLIENT_DATA,       &filer_recursive,
        0);

panel_create_item(ls_flags_panel, PANEL_CYCLE,
        PANEL_ITEM_X,           ATTR_COL(0),
        PANEL_ITEM_Y,           ATTR_ROW(6),
        PANEL_DISPLAY_LEVEL,    PANEL_CURRENT,
        PANEL_LABEL_STRING,      "List '.' files?                            ",
        PANEL_CHOICE_STRINGS,    "No", "Yes", 0,
        PANEL_CLIENT_DATA,       &filer_dot_files,
        0);

panel_create_item(ls_flags_panel, PANEL_CYCLE,
        PANEL_ITEM_X,           ATTR_COL(0),
        PANEL_ITEM_Y,           ATTR_ROW(6),
        PANEL_DISPLAY_LEVEL,    PANEL_CURRENT,
        PANEL_LABEL_STRING,      "Indicate type of file?                    ",
        PANEL_CHOICE_STRINGS,    "No", "Yes", 0,
        PANEL_CLIENT_DATA,       &filer_file_type,

```



```

    0);

done_item = panel_create_item(ls_flags_panel, PANEL_BUTTON,
    PANEL_ITEM_X,          ATTR_COL(1),
    PANEL_ITEM_Y,          ATTR_ROW(8),
    PANEL_LABEL_IMAGE,     panel_button_image(panel, "Done", 6, 0),
    PANEL_NOTIFY_PROC,     done_proc,
    0);

reset_item = panel_create_item(ls_flags_panel, PANEL_BUTTON,
    PANEL_ITEM_X,          ATTR_COL(12),
    PANEL_LABEL_IMAGE,     panel_button_image(panel, "Reset", 6, 0),
    PANEL_NOTIFY_PROC,     reset_proc,
    0);

(void)compose_ls_options(1);
window_fit(ls_flags_panel); /* fit panel around its items */
window_fit(ls_flags_frame); /* fit frame around its panel */
}

char *
compose_ls_options(control)
    int      control;
{
    static char  flags[20];
    char        *ptr;
    char        flag;
    int         first_flag = TRUE;
    Panel_item  item;
    struct Filer *client_data;
    int         index;

    ptr = flags;
    panel_each_item(ls_flags_panel, item)
        if ((item != done_item) && (item != reset_item)) {
            client_data = (struct Filer *)panel_get(item,
                                                    PANEL_CLIENT_DATA, 0);

            switch (control) {
            case 0:
                index = (int)panel_get_value(item);
                defaults_set_integer((char *)client_data->path,
                                    (int)index, (int *)NULL);
                break;
            case 1:
                index = defaults_get_integer((char *)client_data->path,
                                              (int)1, NULL);
                panel_set_value(item, index);
                break;
            case 2:
                flag = client_data->flags[index];
                if (flag != ' ') {
                    if (first_flag) {
                        *ptr++ = '-';
                        first_flag = FALSE;
                    }
                    *ptr++ = flag;
                }
            }
        }
}

```

```

    }
    panel_end_each
    *ptr = '\0';
    return flags;
}

void
ls_proc()
{
    static char previous_dir[MAX_PATH_LEN];
    char *current_dir;
    char cmdstring[100];          /* dir_item's value can be 80, plus flags */

    current_dir = (char *)panel_get_value(dir_item);

    if (strcmp(current_dir, previous_dir)) {
        chdir((char *)panel_get_value(dir_item));
        strcpy(previous_dir, current_dir);
    }
    sprintf(cmdstring, "ls %s %s/%s\n",
            compose_ls_options(),
            current_dir,
            panel_get_value(fname_item));
    ttysw_input(ttysw, cmdstring, strlen(cmdstring));
}

void
ls_flags_proc()
{
    window_set(ls_flags_frame, WIN_SHOW, TRUE, 0);
}

void
done_proc()
{
    window_set(ls_flags_frame, WIN_SHOW, FALSE, 0);
    (void)compose_ls_options(0);
    defaults_write_changed(NULL, NULL);
}

void
reset_proc()
{
    defaults_reread("/Filer", NULL);
}

/* return a pointer to the current selection */
char *
get_selection()
{
    static char    filename[MAX_FILENAME_LEN];
    Seln_holder    holder;
    Seln_request *buffer;

    holder = seln_inquire(SELN_PRIMARY);
    buffer = seln_ask(&holder, SELN_REQ_CONTENTS_ASCII, 0, 0);
    strncpy(
        filename, buffer->data + sizeof(Seln_attribute), MAX_FILENAME_LEN);
}

```



```

    return(filename);
}

/* return 1 if file exists, else print error message and return 0 */
stat_file(filename)
    char *filename;
{
    static char previous_dir[MAX_PATH_LEN];
    char *current_dir;
    char this_file[MAX_PATH_LEN];
    struct stat statbuf;

    current_dir = (char *)panel_get_value(dir_item);

    if (strcmp(current_dir, previous_dir)) {
        chdir((char *)panel_get_value(dir_item));
        strcpy(previous_dir, current_dir);
    }
    sprintf(this_file, "%s/%s", current_dir, filename);
    if (stat(this_file, &statbuf) < 0) {
        char buf[MAX_FILENAME_LEN+11]; /* big enough for message */
        sprintf(buf, "%s not found.", this_file);
        msg(buf, 1);
        return 0;
    }
    return 1;
}

void
edit_proc()
{
    void edit_file_proc(), edit_sel_proc();
    int file_mode = (int)panel_get_value(filing_mode_item);

    if (file_mode) {
        (void)edit_sel_proc();
    } else {
        (void)edit_file_proc();
    }
}

void
edit_file_proc()
{
    char *filename;

    /* return if no selection */
    if (!strlen(filename = (char *)panel_get_value(fname_item))) {
        msg("Please enter a value for \"File:\".", 1);
        return;
    }

    /* return if file not found */
    if (!stat_file(filename))
        return;

    window_set(editsw, TEXTSW_FILE, filename, 0);
}

```

```

    window_set(edit_frame, FRAME_LABEL, filename, WIN_SHOW, TRUE, 0);
}

void
edit_sel_proc()
{
    char *filename;

    /* return if no selection */
    if (!strlen(filename = get_selection())) {
        msg("Please select a file to edit.", 0);
        return;
    }

    /* return if file not found */
    if (!stat_file(filename))
        return;

    window_set(editsw, TEXTSW_FILE, filename, 0);

    window_set(edit_frame, FRAME_LABEL, filename, WIN_SHOW, TRUE, 0);
}

void
del_proc()
{
    char    buf[300];
    char    *filename;
    int     result;
    Event    event;    /* unused */
    int     file_mode = (int)panel_get_value(filing_mode_item);

    /* return if no selection */
    if (file_mode) {
        if (!strlen(filename = get_selection())) {
            msg("Please select a file to delete.", 1);
            return;
        }
    } else {
        if (!strlen(filename = (char *)panel_get_value(fname_item))) {
            msg("Please enter a file name to delete.", 1);
            return;
        }
    }

    /* return if file not found */
    if (!stat_file(filename))
        return;

    /* user must confirm the delete */
    result = alert_prompt(base_frame, &event,
        ALERT_MESSAGE_STRINGS,
        "Ok to delete file:",
        filename,
        0,
        ALERT_BUTTON_YES,    "Confirm, delete file",
        ALERT_BUTTON_NO,    "Cancel",
        0);
}

```



```

switch (result) {
    case ALERT_YES:
        unlink(filename);
        sprintf(buf, "%s deleted.", filename);
        msg(buf, 0);
        break;
    case ALERT_NO:
        break;
    case ALERT_FAILED: /* not likely to happen unless out of FDs */
        printf("Alert failed, will not delete %s.\n", filename);
        break;
}

int
confirm_quit()
{
    int    result;
    Event  event; /* unused */
    char   *msg = "Are you sure you want to Quit?";

    result = alert_prompt(base_frame, &event,
        ALERT_MESSAGE_STRINGS,
        "Are you sure you want to Quit?",
        0,
        ALERT_BUTTON_YES,      "Confirm",
        ALERT_BUTTON_NO,       "Cancel",
        0);
    switch (result) {
        case ALERT_YES:
            break;
        case ALERT_NO:
            return 0;
        case ALERT_FAILED: /* not likely to happen unless out of FDs */
            printf("Alert failed, quitting anyway.\n");
            break;
    }
    return 1;
}

static Notify_value
filer_destroy_func(client, status)
    Notify_client    client;
    Destroy_status    status;
{
    if (status == DESTROY_CHECKING) {
        if (quit_confirmed_from_panel) {
            return(notify_next_destroy_func(client, status));
        } else if (confirm_quit() == 0) {
            (void) notify_veto_destroy((Notify_client) (LINT_CAST(client)));
            return(NOTIFY_DONE);
        }
    }
    return(notify_next_destroy_func(client, status));
}

void
quit_proc()

```

```

{
    if (confirm_quit()) {
        quit_confirmed_from_panel = 1;
        window_destroy(base_frame);
    }
}

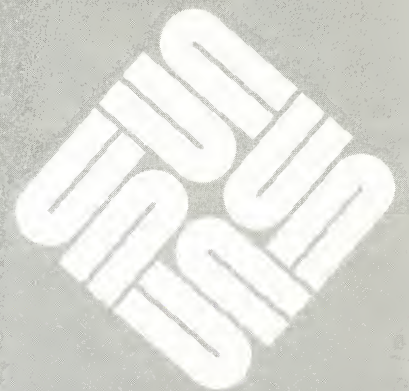
msg(msg, beep)
char *msg;
int    beep;
{
    char    buf[300];
    int     result;
    Event   event; /* unused */
    char    *contine_msg = "Press \"Continue\" to proceed.";

    result = alert_prompt(base_frame, &event,
        ALERT_MESSAGE_STRINGS,
        msg,
        contine_msg,
        0,
        ALERT_NO_BEEPING, (beep) ? 0:1,
        ALERT_BUTTON_YES, "Continue",
        ALERT_TRIGGER,    ACTION_STOP, /* allow either YES or NO answer */
        0);
    switch (result) {
        case ALERT_YES:
        case ALERT_TRIGGERED: /* result of ACTION_STOP trigger */
            break;
        case ALERT_FAILED: /* not likely to happen unless out of FDs */
            printf("Alert failed, continuing anyway.\n");
            break;
    }
}
}

```


Advanced Imaging

Advanced Imaging	173
11.1. Handling Fixup	173
11.2. Icons	174
Loading Icons Dynamically	174
Icon File Format	174
11.3. Damage	176
Handling a SIGWINCH Signal	176
Image Fixup	176
11.4. Pixwin Offset Control	178



Advanced Imaging

The chapter covers some topics on low level image maintenance. There is also a section on icon manipulation.

11.1. Handling Fixup

The routines `pw_read()`, `pw_copy()` and `pw_get()` may find themselves thwarted by trying to read from a portion of the `pixwin` which is hidden, and therefore has no pixels. This can happen with a canvas that you have made non-retained. When this happens, `pw_fixup` (a `struct rectlist`) in the `pixwin` structure will be filled in by the system with the description of the source areas which could not be accessed. The client must then regenerate this part of the image into the destination. Retained `pixwins` will always return `rl_null` in `pw_fixup` because the image is refreshed from the retained memory `pixrect`.

The usual strategy when calling `pw_copy()` is to call the following routine to restrict the `pixwin`'s clipping to just that part of the image that needs to be fixed up.

```
pw_restrict_clipping(pw, rl)
    Pixwin    *pw;
    Rectlist  *rl;
```

You pass in `&pw->pw_fixup` as `rl`. Now you draw your entire `pixwin`. Only the parts that need to be repaired are drawn. Now you need to reset your `pixwin` so that you may access its entire visible surface.

```
pw_exposed(pw)
    Pixwin *pw;
```

`pw_exposed()` is the call that does this.

Dealing with fixup for `pw_read()` or `pw_get()` is really quite ludicrous. One should really run these retained if they are using the screen as a storage medium for their bits.

11.2. Icons

The basic usage of icons is described in the *Icons* chapter of the *SunView Programmer's Guide*. The opaque type `Icon`, and the functions and attributes by which icons are manipulated, are defined in the header file `<suntool/icon.h>`.

Applications such as icon editors or browsers, which need to load icon images at run time, will need to use the functions described in this section. The definitions necessary to use these functions are contained in `<suntool/icon_load.h>`.

Loading Icons Dynamically

You can load an icon's image from a file with the call:

```
int
icon_load(icon, file, error_msg)
    Icon icon;
    char *file, *error_msg;
```

`Icon` is an icon returned by `icon_create()`; `file` is the name of a file created with *iconedit*. `error_msg` is the address of a buffer (at least 256 characters long) into which `icon_load()` will write a message in the event of an error. If `icon_load()` succeeds, it returns zero; otherwise it returns 1.

The function

```
int
icon_init_from_pr(icon, pr)
    Icon icon;
    Pixrect *pr;
```

initializes the width and height of the icon's graphics area (attribute `ICON_IMAGE_RECT`) to match the width and height of `pr`. It also initializes the icon's label (attribute `ICON_LABEL`) to `NULL`. The return value is meaningless.

To load an image from a file into a `pixrect`, use the routine:

```
Pixrect *
icon_load_mpr(file, error_msg)
    char *file, *error_msg;
```

This function allocates a `pixrect`, and loads it with the image contained in `file`. If no problem is encountered, `icon_load_mpr()` returns a pointer to the new `pixrect` containing the image. If it can't access or interpret the file, `icon_load_mpr()` writes a message into the buffer pointed to by `error_msg` and returns `NULL`.

Icon File Format

iconedit writes out an ASCII file consisting of two parts: a comment describing the image, and a list of hexadecimal constants defining the actual pixel values of the image. The contents of the file `<images/template.icon>` are reproduced below, as an example:

```

/* Format_version=1, Width=16, Height=16, Depth=1, Valid_bits_per_item=16
* This file is the template for all images in the cursor/icon library.
* The first line contains the information needed to properly interpret
* the actual bits, which are expected to be used directly by software
* that wants to do compile-time binding to an image via a #include.
* The actual bits must be specified in hex.
* The default interpretation of the bits below is specified by the
* behavior of mpr_static.
* Note that Valid_bits_per_item uses the least-significant bits.
* See also: icon_load.h.
* Description: A cursor that spells "TEMPLATE" using two lines, with a
* solid bar at the bottom.
* Background: White
*/
0xED2F, 0x49E9, 0x4D2F, 0x4928, 0x4D28, 0x0000, 0x0000, 0x8676,
0x8924, 0x8F26, 0x8924, 0xE926, 0x0000, 0x0000, 0xFFFF, 0xFFFF

```

The first line of the comment is composed of header parameters, used by the icon loading routines to properly interpret the actual bits of the image. The `format_version` exists to permit further development of the file format in a compatible manner, and should always be 1. Default values for the other header parameters are `Depth=1`, `Width=64`, `Height=64`, `Valid_bits_per_item=16`.

The remainder of the comment can be used for arbitrary descriptive material.

The following function is provided to allow you to preserve this material when rewriting an image file:

```

FILE *
icon_open_header(file, error_msg, info)
    char          *file, *error_msg;
    icon_header_handle  info;

```

```

typedef struct icon_header_object {
    int    depth,
           height,
           format_version,
           valid_bits_per_item,
           width;
    long   last_param_pos;
} icon_header_object;

```

`icon_open_header` fills in `info` from `file`'s header parameters. `info->last_param_pos` is filled in with the position immediately after the last header parameter that was read. The `FILE *` returned by `icon_open_header()` is left positioned at the end of the header comment. Thus `ftell(icon_open_header())` indicates where the actual bits of the image should begin, and the characters in the range

```
[info->last_param_pos...ftell(icon_open_header())]
```

encompass all of the extra descriptive material contained in the file's header.

11.3. Damage

This section is included for those who can't use the Agent to hide all this complexity. Try to use the Agent, because it is very hard to get the following right.

When a portion of a client's window becomes visible after having been hidden, it is *damaged*. This may arise from several causes. For instance, an overlaying window may have been removed, or the client's window may have been stretched to give it more area. The client is notified that such a region exists either by the events `WIN_REPAINT` or `WIN_RESIZE`, or if the client is not using the Notifier, by the signal `SIGWINCH`; this simply indicates that something about the window has changed in a fashion that probably requires repainting. It is possible that the window has shrunk, and no repainting of the image is required at all, but this is a degenerate case. It is then the client's responsibility to *repair* the damage by painting the appropriate pixels into that area. The following section describes how to do that.

Handling a `SIGWINCH` Signal

There are several stages to handling a `SIGWINCH`. First, in almost all cases, the procedure that catches the signal should *not* immediately try to repair the damage indicated by the signal. Since the signal is a software interrupt, it may easily arrive at an inconvenient time, halfway through a window's repaint for some normal cause, for instance. Consequently, the appropriate action in the signal handler is usually to set a flag which will be tested elsewhere. Conveniently, a `SIGWINCH` is like any other signal; it will break a process out of a `select(2)` system call, so it is possible to awaken a client that was blocked, and with a little investigation, discover the cause of the `SIGWINCH`.

Once a process has discovered that a `SIGWINCH` has occurred and arrived at a state where it's safe to do something about it, it must determine exactly what has changed, and respond appropriately. There are two general possibilities: the window may have changed size, and/or a portion of it may have been uncovered.

`win_getsize()` (described in *Windows*) can be used to inquire the current dimensions of a window. The previous size must have been remembered, for instance from when the window was created or last adjusted. These two sizes are compared to see if the size has changed. Upon noticing that its size has changed, a window containing other windows may wish to rearrange the enclosed windows, for example, by expanding one or more windows to fill a newly opened space.

NOTE *If you are using `window_main_loop()` to drive your program, then the `SIGWINCH` is translated into a `WIN_RESIZE` and/or `WIN_REPAINT` event for you. The rest of this discussion still applies.*

Image Fixup

Whether a size change occurred or not, the actual images on the screen must be fixed up. It is possible to simply repaint the whole window at this point — that will certainly repair any damaged areas — but this is often a bad idea because it typically does much more work than necessary.

Therefore, the window should retrieve the description of the damaged area, repair that damage, and inform the system that it has done so: The `pw_damaged()` procedure:

```
pw_damaged(pw)
    Pixwin *pw;
```

is a procedure much like `pw_exposed()`. It fills in `pwcd_clipping` with a `rectlist` describing the area of interest, stores the id of that `rectlist` in the `pixwin`'s `pw_opshandle` and in `pwcd_damagedid` as well. It also stores its own address in `pwco_getclipping`, so that a subsequent lock will check the correct `rectlist`. All the clippers are set up too. Colormap segment offset initialization is done, as described in *Surface Preparation*.

CAUTION A call to `pw_damaged()` should ALWAYS be made in a `sigwinch` handling routine. Likewise, `pw_donedamaged()` should ALWAYS be called before returning from the `sigwinch` handling routine. While a program that runs on monochrome displays may appear to function correctly if this advice is not followed, running such a program on a color display will produce peculiarities in color appearance.

Now is the time for the client to repaint its window — or at least those portions covered by the damaged `rectlist`; if the regeneration is relatively expensive, that is if the window is large, or its contents complicated, it may be worth restricting the amount of repainting *before* the clipping that the `rectlist` will enforce. This means stepping through the rectangles of the `rectlist`, determining for each what data contributed to its portion of the image, and reconstructing only that portion. See the chapter on `rectlists` for details about *rectlists*.

For retained `pixwins`, the following call can be used to copy the image from the backup `pixrect` to the screen:

```
pw_repairretained(pw)
    Pixwin *pw;
```

When the image is repaired, the client should inform the window system with a call to:

```
pw_donedamaged(pw)
    Pixwin *pw;
```

`pw_donedamaged()` allows the system to discard the `rectlist` describing this damage. It is possible that more damage will have accumulated by this time, and even that some areas will be repainted more than once, but that will be rare.

After calling `pw_donedamaged()`, the `pixwin` describes the entire visible area of the window.

A process which owns more than one window can receive a `SIGWINCH` for any of them, with no indication of which window generated it. The only solution is to fix up *all* windows. Fortunately, that should not be overly expensive, as only the appropriate damaged areas are returned by `pw_damaged()`.

11.4. Pixwin Offset Control

The following routines control the offset of a pixwin's coordinate space. They can be used for writing in a fixed coordinate space even though the pixwin moves about relative to the window's origin.

```
void
pw_set_x_offset(pw, offset)
    Pixwin *pw;
    int      offset;

void
pw_set_y_offset(pw, offset)
    Pixwin *pw;
    int      offset;

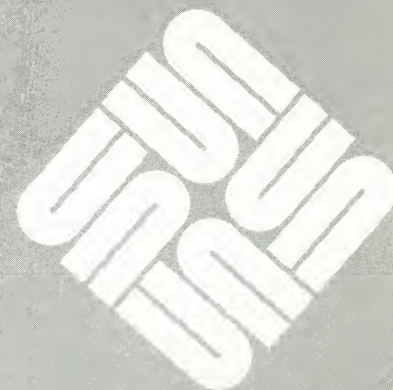
void
pw_set_xy_offset(pw, int x_offset, y_offset)
    Pixwin *pw;
    int      x_offset, y_offset;

int
pw_get_x_offset(pw)
    Pixwin *pw;

int
pw_get_y_offset(pw)
    Pixwin *pw;
```

Menus & Prompts

Menus & Prompts	181
12.1. Full Screen Access	181
Initializing Fullscreen Mode	182
Releasing Fullscreen Mode	182
Seizing All Inputs	182
Grabbing I/O	182
Releasing I/O	182
12.2. Surface Preparation	182
Multiple Plane Groups	183
Pixel Caching	183
Saving Screen Pixels	183
Restoring Screen Pixels	184
Fullscreen Drawing Operations	184



Menus & Prompts

This chapter describes routines that you will probably need when writing a menu or prompt package of your own. Note, however, that SunView's menu and alert facilities documented in the *SunView 1 Programmer's Guide* are both comprehensive full-featured packages that handle fullscreen access for you. Also note that you can use `window_loop()` together with one of the SunView window types to create other kinds of full-screen prompts, again without having to manage fullscreen interaction yourself.

12.1. Full Screen Access

To provide certain kinds of feedback to the user, it may be necessary to violate window boundaries. Pop-up menus, prompts and window management are examples of the kind of operations that do this. The *fullscreen* interface provides a mechanism for gaining access to the entire screen in a safe way. The package provides a convenient interface to underlying *sunwindow* primitives. The following structure is defined in `<suntool/fullscreen.h>`:

```
struct fullscreen {
    int          fs_windowfd;
    struct       rect fs_screenrect;
    struct       pixwin *fs_pixwin;
    struct       cursor fs_cachedcursor;
    struct       inputmask fs_cachedim; /* Pick mask */
    int          fs_cachedinputnext;
    struct       inputmask fs_cachedkbdim; /* Kbd mask */
};
```

`fs_windowfd` is the window that created the `fullscreen` object. `fs_screenrect` describes the entire screen's dimensions. `fs_pixwin` is used to access the screen via the `pixwin` interface. The coordinate space of fullscreen access is the same as `fs_windowfd`'s. Thus, `pixwin` accesses are not necessarily done in the screen's coordinate space. Also, `fs_screenrect` is in the window's coordinate space. If, for example, the screen is 1024 pixels wide and 800 pixels high, `fs_windowfd` has its left edge at 300 and its top edge at 200, that is, both relative to the screen's upper left-hand corner, then `fs_screenrect` is `{-300, -200, 1024, 800}`.

The original cursor, `fs_cachedcursor`, input mask, `fs_cachedim`, and the window number of the input redirection window, `fs_cachedinputnext`, are cached and later restored when the fullscreen access object is destroyed.

Initializing Fullscreen Mode

```
struct fullscreen *
fullscreen_init(windowfd)
    int windowfd;
```

gains full screen access for windowfd and caches the window state that is likely to be changed during the lifetime of the fullscreen object. windowfd is set to do blocking I/O. A pointer to this object is returned.

During the time that the full screen is being accessed, no other processes can access the screen, and all user input is directed to fs->fs_windowfd. Because of this, use fullscreen access infrequently and for only short periods of time.

Releasing Fullscreen Mode

```
fullscreen_destroy(fs)
    struct fullscreen *fs;
```

fullscreen_destroy() restores fs's cached data, releases the right to access the full screen and destroys the fullscreen data object. fs->fs_windowfd's input blocking status is returned to its original state.

Seizing All Inputs

Fullscreen access is built out of the grab I/O mechanism described here. This lower level is useful if you wanted to only grab input.

Normally, input events are directed to the window which underlies the cursor at the time the event occurs (or the window with the keyboard focus, if you have split pick/keyboard focus). Two procedures modify this state of affairs.

Grabbing I/O

A window may temporarily seize all inputs by calling:

```
win_grabio(windowfd)
    int windowfd;
```

The caller's input mask still applies, but it receives input events from the whole screen; no window other than the one identified by windowfd will be offered an input event or allowed to write on the screen after this call.

Releasing I/O

```
win_releaseio(windowfd)
    int windowfd;
```

undoes the effect of a win_grabio(), restoring the previous state.

12.2. Surface Preparation

In order for a client to ignore the offset of his colormap segment the image of the pixwin must be initialized to the value of the offset. This *surface preparation* is done automatically by pixwins under the following circumstances:

- The routine pw_damaged() does surface preparation on the area of the pixwin that is damaged.
- The routine pw_putcolormap() does surface preparation over the entire exposed portion of a pixwin if a new colormap segment is being loaded for the first time.

For monochrome displays, nothing is done during surface preparation. For color displays, when the surface is prepared, the low order bits (colormap segment size

minus 1) are not modified. This means that surface preparation does not clear the image. Initialization of the image (often clearing) is still the responsibility of client code.

There is a case in which surface preparation must be done explicitly by client code. When window boundaries are knowingly violated, as in the case of pop-up menus, the following procedure must be called to prepare each rectangle on the screen that is to be written upon:

```
pw_preparesurface(pw, rect)
    Pixwin *pw;
    Rect *rect;
```

`rect` is relative to `pw`'s coordinate system. Most commonly, a saved copy of the area to be written is made so that it can be restored later — see the next section.

Multiple Plane Groups

On machines with multiple plane groups (such as the Sun-3/110 and other machines using the `cgfour(4S)` frame buffer), `pw_preparesurface()` will correctly set up the enable plane so that the `rect` you are drawing in is visible. If you do not use `pw_preparesurface()`, it is possible that part of the area you are drawing on is displaying values from another plane group, so that part of your image will be occluded.

Pixel Caching

If your application violates window boundaries to put up fullscreen menus and prompts, it is often desirable to remember the state of the screen before you drew on it and then repair it when you are finished. On machines with multiple plane groups such as the Sun-3/110 you need to restore the state of the enable plane and the bits in the other plane group(s). There are routines to help you do this.

Saving Screen Pixels

This routine saves the screen image where you are about to draw:

```
Pw_pixel_cache *
pw_save_pixels(pw, rect);
    Pixwin *pw;
    Rect *rect;

typedef struct pw_pixel_cache {
    Rect rect;
    struct pixrect * plane_group[PIX_MAX_PLANE_GROUPS];
} Pw_pixel_cache;
```

`pw_save_pixels()` tries to allocate memory to store the contents of the pixels in `rect`. If it is unable to, it prints out a message on `stderr` and returns `PW_PIXEL_CACHE_NULL`. If it succeeds, it returns a pointer to a structure which holds the `rect` `rect` and an array of `pixrect`s with the values of the pixels in `rect` in each plane group.

Restoring Screen Pixels

Then, when you have finished fullscreen access, you restore the image which you drew over with:

```
void
pw_restore_pixels(pw, pc);
    Pixwin      *pw;
    Pw_pixel_cache *pc;
```

`pw_restore_pixels()` restores the state of the screen where you drew. All the information it needs is in the `Pw_pixel_cache` pointer that `pw_save_pixels()` returned.

Fullscreen Drawing Operations

If you use `pw_preparesurface()`, you will be given a homogeneous area on which to draw during fullscreen access. However, for applications such as adjusting the size of windows ("rubber-banding"), you do not want to obscure what is underneath. On the other hand, on a machine with multiple plane groups you want your fullscreen access to be visible no matter what plane groups are being displayed.

The following routines perform the same vector drawing, raster operation and `pixwin` copying as their counterparts in *Imaging Facilities: Pixwins* in the *SunView 1 Programmer's Guide*. The difference is that these routines guarantee that the operation will happen in all plane groups so it will definitely be visible on-screen.

CAUTION To save a lot of overhead, these routines make certain assumptions which must be followed.

Anyone calling these `fullscreen_pw_*` routines must

- have called `fullscreen_init()`
- have not done any surface preparation under the pixels affected
- have not called `pw_lock()`
- use the fullscreen `pixwin` during this call
- use a `PIX_NOT(PIX_DST)` operation.

```
void
fullscreen_pw_vector(pw, x0, y0, x1, y1, op, value);
    Pixwin *pw;
    int     x0, y0, x1, y1, op, value;
```

```
void
fullscreen_pw_write(pw, xw, yw, width, height, op,
                   pr, xr, yr);
    Pixwin *pw;
    int     xw, yw, width, height, op, xr, yr;
    Pixrect *pr;
```

```
void
fullscreen_pw_copy(pw, xw, yw, width, height, op,
                  pw_src, xr, yr);
Pixwin *pw, *pw_src;
int     xw, yw, width, height, op, xr, yr;
```


Window Management

Window Management	189
Tool Invocation	190
Utilities	190
13.1. Minimal Repaint Support	192



Window Management

The window management routines provide the standard user interface presented by tool windows:

```

wmgr_open(framefd, rootfd)

wmgr_close(framefd, rootfd)

wmgr_move(framefd)

wmgr_stretch(framefd)

wmgr_top(framefd, rootfd)

wmgr_bottom(framefd, rootfd)

wmgr_refreshwindow(windowfd)

```

`wmgr_open()` opens a frame window from its iconic state to normal size. `wmgr_close()` closes a frame window from its normal size to its iconic size. `wmgr_move()` prompts the user to move a frame window or cancel the operation. If confirmed, the rest of the move interaction, including dragging the window and moving the bits on the screen, is done. `wmgr_stretch()` is like `wmgr_move()`, but it stretches the window instead of moving it. `wmgr_top()` places a frame window on the top of the window stack. `wmgr_bottom()` places the frame window on the bottom of the window stack. `wmgr_refreshwindow()` causes `windowfd` and all its descendant windows to repaint.

The routine `wmgr_changerect()`:

```

wmgr_changerect(feedbackfd, windowfd, event, move, noprompt)
    int      feedbackfd, windowfd;
    Event *event;
    int      move, noprompt;

```

implements `wmgr_move()` and `wmgr_stretch()`, including the user interaction sequence. `windowfd` is moved (1) or stretched (0) depending on the value of `move`. To accomplish the user interaction, the input event is read from the `feedbackfd` window (usually the same as `windowfd`). The prompt is turned off if `noprompt` is 1.


```

int
wmgr_confirm(windowfd, text)
    int    windowfd;
    char *text;

```

`wmgr_confirm()` implements a layer over the prompt package for a standard confirmation user interface. `text` is put up in a prompt box. If the user confirms with a left mouse button press, then `-1` is returned. Otherwise, `0` is returned. The up button event is not consumed.

NOTE *This routine is preserved only for backwards compatibility with versions of the SunOS prior to Release 4.0. You should use the new package documented in the SunView 1 Programmer's Guide for dialogs with the user. `wmgr_confirm()` is not used for user interaction by SunView 1 packages unless the user has disabled Alerts in the Compatibility category of `defaultsedit(1)`.*

Tool Invocation

```

wmgr_forktool(programname, otherargs, rectnormal, recticon,
               iconic)
    char *programname, *otherargs;
    Rect *rectnormal, *recticon;
    int    iconic;

```

is used to fork a new tool that has its normal rectangle set to `rectnormal` and its icon rectangle set to `recticon`. If `iconic` is not zero, the tool is created iconic. `programname` is the name of the file that is to be run and `otherargs` is the command line that you want to pass to the tool. A path search is done to locate the file. Arguments that have embedded white space should be enclosed by double quotes.

Utilities

The utilities described here are some of the low level routines that are used to implement the higher level routines. They may be used to put together a window management user interface different from that provided by tools. If a series of calls is to be made to procedures that manipulate the window tree, the whole sequence should be bracketed by `win_lockdata()` and `win_unlockdata()`, as described in *The Window Hierarchy*.

```

wmgr_completechangerect(windowfd, rectnew, rectoriginal,
                        parentprleft, parentprtop)
    int    windowfd;
    Rect *rectnew, *rectoriginal;
    int    parentprleft, parentprtop;

```

does the work involved with changing the position or size of a window's rect. This involves saving as many bits as possible by copying them on the screen so they don't have to be recomputed. `wmgr_completechangerect()` would be called after some programmatic or user action determined the new window position and size in pixels. `windowfd` is the window being changed. `rectnew` is the window's new rectangle. `rectoriginal` is the window's original rectangle. `parentprleft` and `parentprtop` are the upper-left screen coordinates of the parent of `windowfd`.

```

wmgr_winandchildrenexposed(pixwin, rl)
    Pixwin    *pixwin;
    Rectlist  *rl;

```

computes the visible portion of `pixwin->pw_clipdata.pwcd_windowfd` and its descendants and stores it in `rl`. This is done by any window management routine that is going to try to preserve bits across window changes. For example, `wmgr_completechangerect()` calls `wmgr_winandchildrenexposed()` before and after changing the window size/position. The intersection of the two rectlists from the two calls are those bits that could possibly be saved.

```

wmgr_changelevel(windowfd, parentfd, top)
    int  windowfd, parentfd;
    bool top;

```

moves a window to the top or bottom of the heap of windows that are descendants of its parent. `windowfd` identifies the window to be moved; `parentfd` is the file descriptor of that window's parent, and `top` controls whether the window goes to the top (TRUE) or bottom (FALSE). Unlike `wmgr_top()` and `wmgr_bottom()`, no optimization is performed to reduce the amount of repainting. `wmgr_changelevel()` is used in conjunction with other window rearrangements, which make repainting unlikely. For example, `wmgr_close()` puts the window at the bottom of the window stack after changing its state.

```

#define WMGR_ICONIC WUF_WMGR1

wmgr_iswindowopen(windowfd)
    int  windowfd;

```

The user data of `windowfd` reflects the state of the window via the `WMGR_ICONIC` flag. `WUF_WMGR1` is defined in `<sunwindow/win_ioctl.h>` and `WMGR_ICONIC` is defined in `<suntooll/wmgr.h>`. `wmgr_iswindowopen()` tests the `WMGR_ICONIC` flag and returns TRUE or FALSE as the window is open or closed.

Note that client programs should *never* set or clear the `WMGR_ICONIC` flag.

The `rootfd` window maintains a “next slot” position for both normal tool windows and icon windows in its unused iconic rect data.

`wmgr_setrectalloc()` stores the next slot data and
`wmgr_getrectalloc()` retrieves it:


```

wmgr_setrectalloc(rootfd, tool_left, tool_top,
                  icon_left, icon_top)
int    rootfd;
short  tool_left, tool_top, icon_left, icon_top;

wmgr_getrectalloc(rootfd, tool_left, tool_top,
                  icon_left, icon_top)
int    rootfd;
short *tool_left, *tool_top, *icon_left, *icon_top;

```

If you do a `wmgr_setrectalloc()`, make sure that all the values you are not changing were retrieved with `wmgr_getrectalloc()`. In other words, both procedures affect all the values.

13.1. Minimal Repaint Support

This is an extremely advanced subsection used only for those who might want to implement routines similar of the higher level window management routines mentioned above. This section has strong connections to the *Advanced Imaging* chapter and the chapter on *Rects and Rectlists*. Readers should refer to both from here.

Moving windows about on the screen may involve repainting large portions of their image in new places. Often, the existing image can be copied to the new location, saving the cost of regenerating it. Two procedures are provided to support this function:

```

win_computeclipping(windowfd)
int windowfd;

```

causes the window system to recompute the *exposed* and *damaged* rectlists for the window identified by `windowfd` while withholding the `SIGWINCH` that will tell each owner to repair damage.

```

win_partialrepair(windowfd, r)
int    windowfd;
Rect *r;

```

tells the window system to remove the rectangle `r` from the damaged area for the window identified by `windowfd`. This operation is a no-op if `windowfd` has damage accumulated from a previous window database change, but has not told the window system that it has repaired that damage.

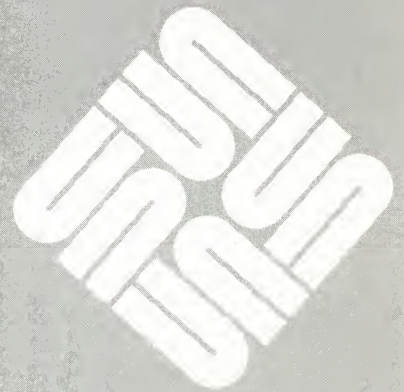
Any window manager can use these facilities according to the following strategy:

- The old exposed areas for the affected windows are retrieved and cached. (`pw_exposed()`)
- The window database is locked and manipulated to accomplish the rearrangement. (`win_lockdata()`, `win_remove()`, `win_setlink()`, `win_setrect()`, `win_insert()`, ...)
- The new area is computed, retrieved, and intersected with the old. (`win_computeclipping()`, `pw_exposed()`, `rl_intersection()`)

- Pixels in the intersection are copied, and those areas are removed from the subject window's damaged area. (`pw_lock()`, `pr_copy()`, `win_partialrepair()`)
- The window database is unlocked, and any windows still damaged get the signals informing them of the reduced damage which must be repaired.

Rects and Rectlists

Rects and Rectlists	197
14.1. Rects	197
Macros on Rects	197
Procedures and External Data for Rects	198
14.2. Rectlists	199
Macros and Constants Defined on Rectlists	200
Procedures and External Data for Rectlists	200



Rects and Rectlists

This chapter describes the geometric structures and operations SunView provides for doing rectangle algebra.

Images are dealt with in rectangular chunks. The basic structure which defines a rectangle is the `Rect`. Where complex shapes are required, they are built up out of groups of rectangles. The structure provided for this purpose is the `Rectlist`.

These structures are defined in the header files `<sunwindow/rect.h>` and `<sunwindow/rectlist.h>`. The library that provides the implementation of the functions of these data types is part of `/usr/lib/libsunwindow.a`.

14.1. Rects

The `rect` is the basic description of a rectangle, and there are macros and procedures to perform common manipulations on a `rect`.

```
#define coord short

typedef struct rect {
    coord    r_left;
    coord    r_top;
    short    r_width;
    short    r_height;
} Rect;
```

The rectangle lies in a coordinate system whose origin is in the upper left-hand corner and whose dimensions are given in pixels.

Macros on Rects

The same header file defines some interesting macros on rectangles. To determine an edge not given explicitly in the `rect`:

```
#define rect_right(rp)
#define rect_bottom(rp)
Rect *rp;
```

returns the coordinate of the last pixel within the rectangle on the right or bottom, respectively.

Useful predicates returning `TRUE` or `FALSE`³² are:

³² `<sunwindow/rect.h>` defines `bool`, `TRUE` and `FALSE` if they are not already defined.

```

rect_isnull(r)                /* r's width or height is 0 */
rect_includespoint(r,x,y)     /* (x,y) lies in r */
rect_equal(r1, r2)           /* r1 and r2 coincide
                             * exactly */
rect_includesrect(r1, r2)     /* every point in r2
                             * lies in r1 */
rect_intersectsrect(r1, r2)   /* at least one point lies
                             * in both r1 and r2 */

Rect *r, *r1, *r2;
coord x, y;

```

Macros which manipulate dimensions of rectangles are:

```

rect_construct(r, x, y, w, h)
Rect *r;
int x, y, w, h;

```

This fills in *r* with the indicated origin and dimensions.

```

rect_marginadjust(r, m)
Rect *r;
int m;

```

adds a margin of *m* pixels on each side of *r*; that is, *r* becomes $2*m$ larger in each dimension.

```

rect_passtoparent(x, y, r)
rect_passtochild(x, y, r)
coord x, y;
Rect *r;

```

sets the origin of the indicated rect to transform it to the coordinate system of a parent or child rectangle, so that its points are now located relative to the parent or child's origin. *x* and *y* are the origin of the parent or child rectangle within *its* parent; these values are added to, or respectively subtracted from, the origin of the rectangle pointed to by *r*, thus transforming the rectangle to the new coordinate system.

Procedures and External Data for Rects

A null rectangle, that is one whose origin and dimensions are all 0, is defined for convenience:

```
extern struct rect rect_null;
```

The following procedures are also defined in `rect.h`:

```

Rect
rect_bounding(r1, r2)
Rect *r1, *r2;

```

This returns the minimal rect that encloses the union of *r1* and *r2*. The returned value is a struct, not a pointer.

```

rect_intersection(r1, r2, rd)
Rect *r1, *r2, *rd;

```

computes the intersection of *r1* and *r2*, and stores that rect into *rd*.


```

bool
rect_clipvector(r, x0, y0, x1, y1)
    Rect *r;
    int  *x0, *y0, *x1, *y1;

```

modifies the vector endpoints so they lie entirely within the rect, and returns FALSE if that excludes the whole vector, otherwise it returns TRUE.

NOTE *This procedure should not be used to clip a vector to multiple abutting rectangles. It may not cross the boundaries smoothly.*

```

bool rect_order(r1, r2, sortorder)
    Rect *r1, *r2;
    int  sortorder;

```

returns TRUE if r1 precedes or equals r2 in the indicated ordering:

```

#define RECTS_TOPTOBOTTOM      0
#define RECTS_BOTTOMTOTOP      1
#define RECTS_LEFTTORIGHT      2
#define RECTS_RIGHTTOLEFT      3

```

Two related defined constants are:

```

#define RECTS_UNSORTED  4

```

indicating a “don’t-care” order, and

```

#define RECTS_SORTS      4

```

giving the number of sort orders available, for use in allocating arrays and so on.

14.2. Rectlists

A **Rectlist** is a structure that defines a list of rects. A number of rectangles may be collected into a list that defines an interesting portion of a larger rectangle. An equivalent way of looking at it is that a large rectangle may be fragmented into a number of smaller rectangles, which together comprise all the larger rectangle’s interesting portions. A typical application of such a list is to define the portions of one rectangle remaining visible when it is partially obscured by others.

```

typedef struct rectlist {
    coord    rl_x, rl_y;
    Rectnode *rl_head;
    Rectnode *rl_tail;
    Rect     rl_bound;
} Rectlist;

typedef struct rectnode {
    Rectnode *rn_next;
    Rect     rn_rect;
} Rectnode;

```

Each node in the rectlist contains a rectangle which covers one part of the visible whole, along with a pointer to the next node. `rl_bound` is the minimal bounding rectangle of the union of all the rectangles in the node list. All rectangles in the rectlist are described in the same coordinate system, which may be translated

efficiently by modifying `rl_x` and `rl_y`.

The routines that manipulate rectlists do their own memory management on rectnodes, creating and freeing them as necessary to adjust the area described by the rectlist.

Macros and Constants Defined on Rectlists

Macros to perform common coordinate transformations are provided:

```
rl_rectoffset(rl, rs, rd)
    Rectlist *rl;
    Rect      *rs, *rd;
```

copies `rs` into `rd`, and then adjusts `rd`'s origin by adding the offsets from `rl`.

```
rl_coordoffset(rl, x, y)
    Rectlist *rl;
    coord     x, y;
```

offsets `x` and `y` by the offsets in `rl`. For instance, it converts a point in one of the rects in the rectnode list of a rectlist to the coordinate system of the rectlist's parent.

Parallel to the macros on rect's, we have:

```
rl_passtoparent(x, y, rl)
rl_passtochild(x, y, rl)
    coord      x, y;
    Rectlist *rl;
```

which add or subtract the given coordinates from the rectlist's `rl_x` and `rl_y` to convert the `rl` into its parent's or child's coordinate system.

Procedures and External Data for Rectlists

An empty rectlist is defined, which should be used to initialize any rectlist before it is operated on:

```
extern struct rectlist rl_null;
```

Procedures are provided for useful predicates and manipulations. The following declarations apply uniformly in the descriptions below:

```
Rectlist *rl, *rl1, *rl2, *rld;
Rect      *r;
coord     x, y;
```

Predicates return TRUE or FALSE. Refer to the following table for specifics.

Table 14-1 *Rectlist Predicates*

<i>Macro</i>	<i>Returns TRUE if</i>
<code>rl_empty(rl)</code>	Contains only null rects
<code>rl_equal(rl1, rl2)</code>	The two rectlists describe the same space identically — same fragments in the same order
<code>rl_includespoint(rl,x,y)</code>	(x,y) lies within some rect of rl
<code>rl_equalrect(r, rl)</code>	rl has exactly one rect, which is the same as r
<code>rl_boundintersectsrect(r, rl)</code>	Some point lies both in r and in rl's bounding rect

Manipulation procedures operate through side-effects, rather than returning a value. Note that it is legitimate to use a rectlist as both a source and destination in one of these procedures. The source node list will be freed and reallocated appropriately for the result. Refer to the following table for specifics.

Table 14-2 *Rectlist Procedures*

<i>Procedure</i>	<i>Effect</i>
<code>rl_intersection(r11, r12, rld)</code>	Stores into <code>rld</code> a rectlist which covers the intersection of <code>r11</code> and <code>r12</code> .
<code>rl_union(r11, r12, rld)</code>	Stores into <code>rld</code> a rectlist which covers the union of <code>r11</code> and <code>r12</code> .
<code>rl_difference(r11, r12, rld)</code>	Stores into <code>rld</code> a rectlist which covers the area of <code>r11</code> not covered by <code>r12</code> .
<code>rl_coalesce(rl)</code>	Attempts to shorten <code>rl</code> by coalescing some of its fragments. An <code>rl</code> whose bounding rect is completely covered by the union of its node rects will be collapsed to a single node; other simple reductions will be found; but the general solution to the problem is not attempted.
<code>rl_sort(rl, rld, sort)</code> <code>int sort;</code>	<code>rl</code> is copied into <code>rld</code> , with the node rects arranged in sort order.
<code>rl_rectintersection(r, rl, rld)</code>	<code>rld</code> is filled with a rectlist that covers the intersection of <code>r</code> and <code>rl</code> .
<code>rl_rectunion(r, rl, rld)</code>	<code>rld</code> is filled with a rectlist that covers the union of <code>r</code> and <code>rl</code> .
<code>rl_rectdifference(r, rl, rld)</code>	<code>rld</code> is filled with a rectlist that covers the portion of <code>rl</code> which is not in <code>r</code> .
<code>rl_initwithrect(r, rl)</code>	Fills in <code>rl</code> so that it covers the rect <code>r</code> .
<code>rl_copy(rl, rld)</code>	Fills in <code>rld</code> with a copy of <code>rl</code> .
<code>rl_free(rl)</code>	Frees the storage allocated to <code>rl</code> .
<code>rl_normalize(rl)</code>	Resets <code>rl</code> 's offsets (<code>rl_x</code> , <code>rl_y</code>) to be 0 after adjusting the origins of all rects in <code>rl</code> accordingly.

Scrollbars

Scrollbars	205
15.1. Basic Scrollbar Management	205
Registering as a Scrollbar Client	205
Keeping the Scrollbar Informed	206
Handling the <code>SCROLL_REQUEST</code> Event	207
Performing the Scroll	208
Normalizing the Scroll	209
Painting Scrollbars	209
15.2. Advanced Use of Scrollbars	209
Types of Scrolling Motion in Simple Mode	210
Types of Scrolling Motion in Advanced Mode	211



Scrollbars

Canvases, text subwindows and panels have been designed to work with scrollbars. The text subwindow automatically creates its own vertical scrollbar. For canvases and panels, it is your responsibility to create the scrollbar and pass it in via the attributes `WIN_VERTICAL_SCROLLBAR` or `WIN_HORIZONTAL_SCROLLBAR`.

The chapter on scrollbars in the *SunView Programmer's Guide* covers what most applications need to know about scrollbars.

The material in this chapter will be of interest only if you are writing an application not based on canvases, text subwindows or panels, and you need to communicate with the scrollbar directly as events are received. This chapter is directed to programmers writing software which receives scroll-request events and implements scrolling.

The definitions necessary to use scrollbars are found in the header file `<suntool/scrollbar.h>`

15.1. Basic Scrollbar Management

Registering as a Scrollbar Client

The scrollbar receives events directly from the Notifier. The user makes a scroll request by releasing a mouse button over the scrollbar. The scrollbar's job is to translate the button-up event into a scrolling request event, and send this event, via the Notifier, to its client.

To receive scrolling request events, the client must register itself with the scrollbar via the `SCROLL_NOTIFY_CLIENT` attribute. For example, `panel_1` would register as a client of `bar_1` with the call:

```
scrollbar_set(bar_1, SCROLL_NOTIFY_CLIENT, panel_1, 0);
```

NOTE *Before registering with the scrollbar, the client must register with the Notifier by calling `win_register()`.*

In most applications, such as the panel example above, the client and the scrolling object are identical. However, they may well be distinct. In such a case, if the client wants the scrollbar to keep track of which object the scrollbar is being used with, the client has to inform the scrollbar explicitly of the object which is

to be scrolled. This is done by setting the `SCROLL_OBJECT` attribute.

For example, in the text subwindow package, the text subwindow is the client to be notified. Within a given text subwindow there may be many views onto the underlying file. Each of these views has its own scrollbar. So each scrollbar created by the text subwindow will have the text subwindow as `SCROLL_NOTIFY_CLIENT` and the particular view as `SCROLL_OBJECT`. So to create scrollbars for two views, the text subwindow package would call:

```
scrollbar_set(bar_1,
              SCROLL_NOTIFY_CLIENT, textsubwindow_1,
              SCROLL_OBJECT,        view_1,
              0);

scrollbar_set(bar_2,
              SCROLL_NOTIFY_CLIENT, textsubwindow_1,
              SCROLL_OBJECT,        view_2,
              0);
```

Keeping the Scrollbar Informed

The visible portion of the scrolling object is called the *view* into the object. The scrollbar displays a *bubble* representing both the location of the view within the object and the size of the view relative to the size of the object. In order to compute the size and location of the bubble, and to compute the new offset into the object after a scroll, the scrollbar needs to know the current lengths of both the object and the view.

The client must keep the scrollbar informed by setting the attributes `SCROLL_OBJECT_LENGTH` and `SCROLL_VIEW_LENGTH`. There are two obvious strategies for when to update this information. You can ensure that the scrollbar is always up-to-date by informing it whenever the lengths in question change. If this is too expensive (because the lengths change too frequently) you can update the scrollbar only when the cursor enters the scrollbar.

This strategy of updating the scrollbar when it is entered can be implemented as follows. When the scrollbar gets a `LOC_RGNENTER` or `LOC_RGNEXIT` event, it causes the event-handling procedure of its notify client to be called with a `SCROLL_ENTER` or `SCROLL_EXIT` event. The client then catches the `SCROLL_ENTER` event and updates the scrollbar, as in the example below. (Note that the scrollbar handle to use for the `scrollbar_set()` call is passed in as `arg`).


```

Notify_value
panel_event_proc(client, event, arg, type)
    caddr_t      client;
    Event        *event;
    Notify_arg    arg;
    Notify_event_type type;
{
    switch (event_id(event)) {
        ...
        case SCROLL_ENTER:
            scrollbar_set((Scrollbar) arg,
                SCROLL_OBJECT_LENGTH, current_obj_length,
                SCROLL_VIEW_START,   current_view_start,
                SCROLL_VIEW_LENGTH,   current_view_length,
                0);
            break;
        ...
    }
}

```

The client can interpret the values of `SCROLL_OBJECT_LENGTH`, `SCROLL_VIEW_LENGTH` and `SCROLL_VIEW_START` in whatever units it wants to. For example, the panel package uses pixel units, while the text subwindow package uses character units.

Handling the SCROLL_REQUEST Event

When the user requests a scroll, the scrollbar client's event-handling procedure gets called with an event whose event code is `SCROLL_REQUEST`. The event proc is passed an argument (`arg` in the example below) for event-specific data. In the case of scrollbar-related events, `arg` is a scrollbar handle (type `Scrollbar`). As in the example below, the client's event proc must switch on the `SCROLL_REQUEST` event and call a procedure to actually perform the scroll:

```

Notify_value
panel_event_proc(panel, event, arg, type)
    Panel        panel;
    Event        *event;
    Notify_arg    arg;
    Notify_event_type type;
{
    switch (event_id(event)) {
        ...
        case SCROLL_REQUEST:
            do_scroll(panel, (Scrollbar) arg);
            break;
        ...
    }
}

```

Performing the Scroll

The new offset into the scrolling object is computed by the Scrollbar Package, and is available to the client via the attribute `SCROLL_VIEW_START`. The client's job is to paint the object starting at the new offset, and to paint the scrollbar reporting the scroll, so that its bubble will be updated to reflect the new offset. So in the simplest case the client's scrolling routine would look something like this:

```
do_scroll(sb)
    Scrollbar sb;
{
    unsigned new_view_start;

    /* paint scrollbar to show bubble in new position */
    scrollbar_paint(sb);

    /* get new offset into object from scrollbar */
    new_view_start = (unsigned) scrollbar_get(sb,
                                           SCROLL_VIEW_START);

    /* client's proc to paint object at new offset */
    paint_object(sb->object, new_view_start);
}
```

If the client has both a horizontal and a vertical scrollbar, it will probably be necessary to distinguish the direction of the scroll, as in:

```
do_scroll(sb)
    Scrollbar sb;
{
    /* paint the scrollbar to show bubble in new position */
    scrollbar_paint(sb);

    /* get new offset into object from scrollbar */

    /*
     * pass the new offset and the direction of the scrollbar
     * into the paint function
     */
    paint_object(sb->object,
                scrollbar_get(sb, SCROLL_VIEW_START),
                scrollbar_get(sb, SCROLL_DIRECTION));
}
```

In order to repaint the screen efficiently, you need to know which bits appear on the screen both before and after the scroll, and thus can be copied to their new location with `pw_copy()`. To compute the copyable region you will need, in addition to the current offset into the object (`SCROLL_VIEW_START`), the offset prior to the scroll (`SCROLL_LAST_VIEW_START`).

Note: you are responsible for repainting the scrollbar after a scroll, with one of the routines described later in *Painting Scrollbars*.

Normalizing the Scroll

The scrollbar package can be utilized in two modes: *normalized* and *un-normalized*. Un-normalized means that when the user makes a scrolling request, it is honored exactly to the pixel, as precisely as resolution permits. In normalized scrolling, the client makes an attempt to put the display in some kind of "normal form" after the scrolling has taken place.

To take panels as an example, this simply means that after a vertical scroll, the Panel Package modifies the value of `SCROLL_VIEW_START` so that the highest item which is either fully or partially visible in the panel is placed with its top edge `SCROLL_MARGIN` pixels from the top of the panel.

Normalization is enabled by setting the `SCROLL_NORMALIZE` attribute for the scrollbar to `TRUE`, and the `SCROLL_MARGIN` attribute to the desired margin. `SCROLL_NORMALIZE` defaults to `TRUE`, and `SCROLL_MARGIN` defaults to four pixels.

Note that the scrollbar package simply keeps track of whether the scrolls it computes are intended to be normalized or not. The client who receives the scroll-request event is responsible for asking the scrollbar whether normalization should be done, and if so, doing it.

After the client computes the normalized offset, it must update the scrollbar by setting the attribute `SCROLL_VIEW_START` to the normalized offset.

Painting Scrollbars

The basic routine to paint a scrollbar is:

```
scrollbar_paint(scrollbar);
Scrollbar scrollbar;
```

`scrollbar_paint()` repaints only those portions of the scrollbar (page buttons, bar proper, and bubble) which have been modified since they were last painted. To clear and repaint all portions of the bar, use `scrollbar_paint_clear()`.

In addition, the routines `scrollbar_paint_bubble()` and `scrollbar_clear_bubble()` are provided to paint or clear the bubble only.

15.2. Advanced Use of Scrollbars

As indicated previously under *Performing the Scroll*, the client need not be concerned with the details of the scroll request at all — he may simply use the new offset given by the value of the `SCROLL_VIEW_START` attribute. However, the client may want to assume partial or full responsibility for the scroll. He may compute the new offset from scratch himself, or start with the offset computed by the Scrollbar Package and modify it so as not to have text or other information clipped at the top of the window (see the preceding discussion under *Normalizing the Scroll*).

In order to give you complete control over the scroll, attributes are provided to allow you to retrieve all the information about the scroll-request event and the object's state at the time of the event. The attributes of interest include:

Table 15-1 *Scroll-Related Scrollbar Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
SCROLL_LAST_VIEW_START	int	Offset of view into object prior to scroll. Get only.
SCROLL_OBJECT	caddr_t	pointer to the scrollable object.
SCROLL_OBJECT_LENGTH	int	Length of scrollable object, in client units (value must be ≥ 0). Default: 0.
SCROLL_REQUEST_MOTION	Scroll_motion	Scrolling motion requested by user.
SCROLL_REQUEST_OFFSET	int	Pixel offset of scrolling request into scrollbar. Default: 0.
SCROLL_VIEW_LENGTH	int	Length of viewing window, in client units. Default: 0.
SCROLL_VIEW_START	int	Current offset into scrollable object, measured in client units. (Value must be ≥ 0). Default: 0.

Types of Scrolling Motion in Simple Mode

There are three basic types of scrolling motion:

- SCROLL_ABSOLUTE. This is the "thumbing" motion requested by the user with the middle button. You can retrieve the number of pixels into the scrollbar of the request (including the page button which may be present) via SCROLL_REQUEST_OFFSET.
- SCROLL_FORWARD. This is to be interpreted as a request to bring the location of the cursor to the top (left, if horizontal).
- SCROLL_BACKWARD. This is to be interpreted as a request to bring the top (left, if horizontal) point to the cursor.

The function which implements scrolling may want to switch on the scrolling motion, to implement different algorithms for each motion. In the following example, `do_absolute_scroll()`, `do_forward_scroll()`, `do_backward_scroll()` and `paint_object()` are procedures written by the client:

```

do_scroll(sb)
    Scrollbar sb;
{
    unsigned new_offset;
    Scroll_motion motion;

    motion = (Scroll_motion) scrollbar_get(sb, SCROLL_MOTION);

    switch (motion) {

        case SCROLL_ABSOLUTE:
            new_offset = do_absolute_scroll(sb);
            break;

        case SCROLL_FORWARD:
            new_offset = do_forward_scroll(sb);
            break;

        case SCROLL_BACKWARD:
            new_offset = do_backward_scroll(sb);
            break;
    }

    /* tell the scrollbar of the new offset */
    scrollbar_set(sb, SCROLL_VIEW_START, new_offset, 0);

    /* paint scrollbar to show bubble in new position */
    scrollbar_paint(sb);

    /* client's repainting proc */
    paint_object(scrollbar_get(sb, SCROLL_OBJECT, 0);
}

```

Types of Scrolling Motion in Advanced Mode

Internally, the scrollbar package distinguishes nine different types of motion, depending on which mouse button the user pressed, the state of the shift key, and the whether the cursor was in the bar, the forward page button or the backward page button. Normally, these motions are mapped onto the three basic motions described above. In order to perform this mapping, the scrollbar package needs to know the distance between lines. You do this by setting the `SCROLL_LINE_HEIGHT` attribute, as in:

```
scrollbar_set(sb, SCROLL_LINE_HEIGHT, 20, 0);
```

NOTE *This is the distance, in pixels, from the top of one line to the top of the succeeding line.*

The scrollbar package can also be used in *advanced mode*, in which case the mapping described above is not performed -- the motion is passed as is to the notify proc. This allows you to interpret each motion exactly as you want.

The following table gives the nine motions, the user action which generates them, and the basic motions which they are mapped onto (if not in advanced

mode):

Table 15-2 *Scrollbar Motions*

<i>Motion</i>	<i>Generated By</i>	<i>Mapped to</i>
ABSOLUTE	middle in bar	ABSOLUTE
POINT_TO_MIN	left in bar	FORWARD
MAX_TO_POINT	shifted left in bar	FORWARD
PAGE_FORWARD	middle in page button	FORWARD
LINE_FORWARD	left in page button	FORWARD
MIN_TO_POINT	right in bar	BACKWARD
POINT_TO_MAX	shifted right in bar	BACKWARD
PAGE_BACKWARD	shifted middle in page button	BACKWARD
LINE_BACKWARD	right in page button	BACKWARD

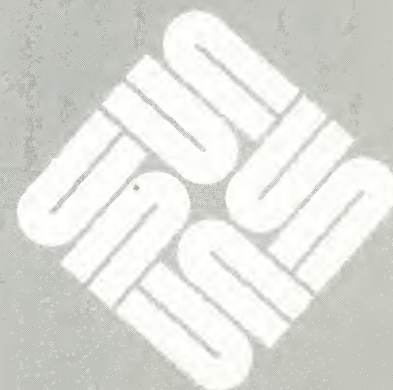
To operate in advanced mode you must:

- set the attribute `SCROLL_ADVANCED_MODE` to `TRUE`.
- Switch on the nine motions in the above table. Note: specifically, `SCROLL_FORWARD` and `SCROLL_BACKWARD` must *not* appear in your switch statement. In other words, for basic mode switch on the three basic motions; for advanced mode switch on the nine advanced motions.

A

Writing a Virtual User Input Device Driver

Writing a Virtual User Input Device Driver	215
A.1. Firm Events	215
The Firm_event Structure	215
Pairs	216
Choosing VUID Events	217
A.2. Device Controls	217
Output Mode	217
Device Instancing	217
Input Controls	218
A.3. Example	218



Writing a Virtual User Input Device Driver

This section describes what a device driver needs to do in order to conform to the Virtual User Input Device (vuid) interface understood by SunView. This is not a tutorial on writing a device driver; only the vuid related aspects of device driver writing are covered.

A.1. Firm Events

A stream of *firm events* is what your driver is expected to emit when called through the `read()` system call. This stream is simply a byte stream that encodes `Firm_event` structures. A firm event is a structure comprising an ID which indicates what kind of event it is, the value of the event, and a time when this event occurred; it also carries some information that allows the event's eventual consumer to maintain the complete state of its input system.

The `Firm_event` Structure

The `Firm_event` structure is defined in `<sundev/vuid_event.h>`:

```
typedef struct firm_event {
    u_short      id;
    u_char       pair_type;
    u_char       pair;
    int          value;
    struct timeval time;
} Firm_event;

#define FE_PAIR_NONE      0
#define FE_PAIR_SET       1
#define FE_PAIR_DELTA     2
#define FE_PAIR_ABSOLUTE  3
```

Here is what the fields in the `Firm_event` mean:

- `id` — is the event's unique identifier. It is either the `id` of an existing vuid event (if you're trying to emulate part of the vuid) or your one of your own creation (see *Choosing VUID Events*).
- `value` — is the event's value. It is often 0 (up) or 1 (down). For valuators it is a 32 bit integer.
- `time` — is the event's time stamp, i.e., when it occurred. The time stamp is not defined to be meaningful except to compare with other `Firm_event` time stamps. In the kernel, a call to `uniqtime()`, which takes a pointer to a `struct timeval`, gets you a close-to-current unique time. In user

process land, a call to `gettimeofday(2)` gets time from the same source (but it is not made unique).

Pairs

This brings us to `pair_type` and `pair`. These two fields enable a consumer of events to maintain input state in an event-independent way. The `pair` field is critical for a input state maintenance package, one that is designed to not know anything about the semantics of particular events, to maintain correct data for corresponding absolute, delta and paired state variables. Some examples will make this clear:

- Say you have a tablet emitting absolute locations. Depending on the client, what the absolute location is may be important (say for digitizing) and then again the difference between the current location and the previous location may be of interest (say for computing acceleration while tracking a cursor).
- Say you are keyboard in which the user has typed `^C`. Your driver first emits a `SHIFT_CTRL` event as the control key goes down. Next your driver emits a `^C` event (one of the events from the ASCII void segment) as the `c` key goes down. Now the application that you are driving happens to be using the `c` key as a shift key in some specialized application. The application wants to be able to say to SunView (the maintainer of the input state), "Is the `c` key down?" and get a correct response.

The void supports a notion of updating a companion event at the same time that a single event is generated. In the first situations above, the tablet wants to be able to update companion absolute and relative event values with a single event. In the second situations above, the keyboard wants to be able to update companion `^C` and `c` event values with a single event. The void supports this notion of updating a companion event in such a way as to be independent from these two particular cases. `pair_type` defines the type of the companion event:

- `FE_PAIR_NONE` — is the common case in which `pair` is not defined, i.e., there is no companion.
- `FE_PAIR_SET` — is used for ASCII controlled events in which `pair` is the uncontrolled *base* event, e.g., `^C` and `'c'` or `'C'`, depending on the state of the shift key. The use of this pair type is not restricted to ASCII situations. This pair type simply says to set the *pairth* event in *id*'s void segment to be *value*.
- `FE_PAIR_DELTA` — identifies `pair` as the delta companion to *id*. This means that the *pairth* event in *id*'s void segment should be set to the delta of *id*'s current value and *value*. One should always create void valuator events as delta/absolute pairs. For example, the events `LOC_X_DELTA` and `LOC_X_ABSOLUTE` are pairs and the events `LOC_Y_DELTA` and `LOC_Y_ABSOLUTE` are pairs. These events are part of the standard `WORKSTATION_DEVID` segment that define *the* distinguish primary locator motion events.
- `FE_PAIR_ABSOLUTE` — identifies `pair` as the absolute companion to *id*. This means that the *pairth* event in *id*'s void segment should be set to the sum of *id*'s current value and *value*. One should always create void

valuator events as delta/absolute pairs.

As indicated by the previous discussion, `pair` must be in the same void segment as `id`.

Choosing VUID Events

One needs to decide which events the driver is going to emit. If you want to emulate the Sun virtual workstation then you want to emit the same events as the `WORKSTATION_DEVID` void segment. A tablet, for example, can emit absolute locator positions `LOC_X_ABSOLUTE` and `LOC_Y_ABSOLUTE`, instead of a mouse's relative locator motions `LOC_X_DELTA` and `LOC_Y_DELTA`. SunView will use these to drive the mouse.

If you have a completely new device then you want to create a new void segment. This is talked about in the workstations chapter of the *SunView System Programmer's Guide*.

A.2. Device Controls

A void driver is expected to respond to a variety of device controls.

Output Mode

Many of you will be starting from an existing device driver that already speaks its own native protocol. You may not want to flush this old protocol in favor of the void protocol. In this case you may want to operate in both modes. `VUID*FORMAT` ioctls are used to control which byte stream format that an input device should emit.

```
#define VUIDSFORMAT    _IOW(v, 1, int)
#define VUIDGFORMAT    _IOR(v, 2, int)

#define VUID_NATIVE    0
#define VUID_FIRM_EVENT 1
```

`VUIDSFORMAT` sets the input device byte stream format to one of:

- `VUID_NATIVE` — The device's native byte stream format (it may be void).
- `VUID_FIRM_EVENT` — The byte stream format is `Firm_events`.

An error of `ENOTTY` or `EINVAL` indicates that a device can't speak `Firm_events`.

`VUIDSFORMAT` gets the input device byte stream format.

Device Instancing

`VUID*ADDR` ioctls are used to control which address a particular virtual user input device segment has. This is used to have an instancing capability, e.g., a second mouse. One would:

- Take the current mouse driver, which emits events in the `WORKSTATION_DEVID` void segment.
- Define a new void segment, say `LOC2_DEVID`.
- Add `LOC2_X_ABSOLUTE`, `LOC2_Y_ABSOLUTE`, `LOC2_X_DELTA` and `LOC2_Y_DELTA` to the `LOC2_DEVID` void segment at the same offset from the beginning of the segment as `LOC_X_ABSOLUTE`, `LOC_Y_ABSOLUTE`, `LOC_X_DELTA` and `LOC_Y_DELTA` in the `WORKSTATION_DEVID`.

- Command a mouse to emit events using LOC2_DEVID's segment address and the mouse's original low byte segment offsets. Thus, it would be emitting LOC2_X_DELTA and LOC2_Y_DELTA, which is what your application would eventually receive.

Here is the VUID*ADDR commands common data structure and command definitions:

```
typedef struct    void_addr_probe {
    short        base;
    union        {
        short    next;
        short    current;
    } data;
} Vuid_addr_probe;

#define VUIDSADDR    _IOW(v, 3, struct    void_addr_probe)
#define VUIDGADDR    _IOWR(v, 4, struct    void_addr_probe)
```

VUIDSADDR is used to set an alternative void segment. `base` is the void device addr that you are changing. A void device addr is the void segment id shifted into it's high byte position. `data.next` is the new void device addr that should be used instead of `base`. An `errno` of `ENOTTY` indicates that a device can't deal with these commands. An `errno` of `ENODEV` indicates that the requested virtual device has no events generated for it by this physical device.

VUIDGADDR is used to get an current value of a void segment. `base` is the default void device addr that you are asking about. `data.current` is the current void device addr that is being used instead of `base`.

The implementation of these `ioctl`s is optional. If you don't do it then your device wouldn't be able to support multiple instances.

Input Controls

Your device needs to support non-blocking reads in order to run with SunView 3.0. This means that the `read(2)` system call returns `EWOULDBLOCK` when no input is available.

In addition, your driver should support the `select(2)` system call and asynchronous input notification (sending `SIGIO` when input pending). However, your driver will still run without these two things in 3.0 SunView.

A.3. Example

The following example is parts of code taken from the Sun 3.0 mouse driver. It illustrates some of the points made above.

```
/* Copyright (c) 1985 by Sun Microsystems, Inc. */

<elided material>

#include "../sundev/vuid_event.h"

/*
 * Mouse select management is done by utilizing the tty mechanism.
 * We place a single character on the tty raw input queue whenever
```



```

* there is some amount of mouse data available to be read. Once,
* all the data has been read, the tty raw input queue is flushed.
*
* Note: It is done in order to get around the fact that line
* disciplines don't have select operations because they are always
* expected to be ttys that stuff characters when they get them onto
* a queue.
*
* Note: We use spl5 for the mouse because it is functionally the
* same as spl6 and the tty mechanism is using spl5. The original
* code that was doing its own select processing was using spl6.
*/
#define spl_ms spl5

/* Software mouse registers */
struct ms_softc {
    struct mousebuf {
        short    mb_size;           /* size (in mouseinfo units) of buf */
        short    mb_off;           /* current offset in buffer */
        struct mouseinfo {
            char    mi_x, mi_y;
            char    mi_buttons;
#define MS_HW_BUT1    0x4    /* left button position */
#define MS_HW_BUT2    0x2    /* middle button position */
#define MS_HW_BUT3    0x1    /* right button position */
            struct timeval mi_time; /* timestamp */
        } mb_info[1];           /* however many samples */
    } *ms_buf;
    short    ms_bufbytes;           /* buffer size (in bytes) */
    short    ms_flags;             /* currently unused */
    short    ms_oldoff;            /* index into mousebuf */
    short    ms_oldoff1;           /* at mi_x, mi_y or mi_buttons... */
    short    ms_readformat;        /* format of read stream */
#define MS_3BYTE_FORMAT VUID_NATIVE    /* 3 byte format (buts/x/y) */
#define MS_VUID_FORMAT VUID_FIRM_EVENT /* void Firm_event format */
    short    ms_vuidaddr;          /* void addr for MS_VUID_FORMAT */
    short    ms_vuidcount;         /* count of unread firm events */
    short    ms_samplecount;       /* count of unread mouseinfo samples */
    char    ms_readbuttons;        /* button state as of last read */
};

struct msdata {
    struct ms_softc msd_softc;
    struct tty *msd_tp;
};

<elided material>

};
struct msdata msdata[NMS];
struct msdata *mstptomsd();

<elided material>

```

```

/* Open a mouse.  Calls sets mouse line characteristics */
/* ARGSUSED */
msopen(dev, tp)
    dev_t dev;
    struct tty *tp;
{
    register int err, i;
    struct sgttyb sg;
    register struct mousebuf *b;
    register struct ms_softc *ms;
    register struct msdata *msd;
    caddr_t zmemall();
    register struct cdevsw *dp;

    /* See if tp is being used to drive ms already.  */
    for (i = 0; i < NMS; ++i)
        if (msdata[i].msd_tp == tp)
            return(0);
    /* Get next free msdata */
    for (i = 0; i < NMS; ++i)
        if (msdata[i].msd_tp == 0)
            goto found;
    return(EBUSY);
found:
    /* Open tty */
    if (err = ttyopen(dev, tp))
        return(err);
    /* Setup tty flags */
    dp = &cdevsw[major(dev)];
    if (err = (*dp->d_ioctl) (dev, TIOCGETP, (caddr_t)&sg, 0))
        goto error;
    sg.sg_flags = RAW+ANYP;
    sg.sg_ispeed = sg.sg_ospeed = B1200;
    if (err = (*dp->d_ioctl) (dev, TIOCSETP, (caddr_t)&sg, 0))
        goto error;
    /* Set up private data */
    msd = &msdata[i];
    msd->msd_xnext = 1;
    msd->msd_tp = tp;
    ms = &msd->msd_softc;
    /* Allocate buffer and initialize data */
    if (ms->ms_buf == 0) {
        ms->ms_bufbytes = MS_BUF_BYTES;
        b = (struct mousebuf *)zmemall(memall, ms->ms_bufbytes);
        if (b == 0) {
            err = EINVAL;
            goto error;
        }
        b->mb_size = 1 + (ms->ms_bufbytes - sizeof (struct mousebuf))
            / sizeof (struct mouseinfo);
        ms->ms_buf = b;
        ms->ms_vuidaddr = VKEY_FIRST;
        msflush(msd);
    }
}

```

```

    }
    return (0);
error:
    bzero((caddr_t)msd, sizeof (*msd));
    bzero((caddr_t)ms, sizeof (*ms));
    return (err);
}

/*
 * Close the mouse
 */
msclose(tp)
    struct tty *tp;
{
    register struct msdata *msd = mstptomsd(tp);
    register struct ms_softc *ms;

    if (msd == 0)
        return;
    ms = &msd->msd_softc;
    /* Free mouse buffer */
    if (ms->ms_buf != NULL)
        wmemfree((caddr_t)ms->ms_buf, ms->ms_bufbytes);
    /* Close tty */
    ttyclose(tp);
    /* Zero structures */
    bzero((caddr_t)msd, sizeof (*msd));
    bzero((caddr_t)ms, sizeof (*ms));
}

/*
 * Read from the mouse buffer
 */
msread(tp, uio)
    struct tty *tp;
    struct uio *uio;
{
    register struct msdata *msd = mstptomsd(tp);
    register struct ms_softc *ms;
    register struct mousebuf *b;
    register struct mouseinfo *mi;
    register int error = 0, pri, send_event, hwbit;
    register char c;
    Firm_event fe;

    if (msd == 0)
        return(EINVAL);
    ms = &msd->msd_softc;
    b = ms->ms_buf;
    pri = spl_ms();
    /*
     * Wait on tty raw queue if this queue is empty since the tty is
     * controlling the select/wakeup/sleep stuff.
    */

```



```

    */
    while (tp->t_rawq.c_cc <= 0) {
        if (tp->t_state & TS_NBIO) {
            (void) splx(pri);
            return (EWOULDBLOCK);
        }
        sleep((caddr_t)&tp->t_rawq, TTIPRI);
    }
    while (!error && (ms->ms_oldoff1 || ms->ms_oldoff != b->mb_off)) {
        mi = &b->mb_info[ms->ms_oldoff];
        switch (ms->ms_readformat) {

            case MS_3BYTE_FORMAT:
                break;

            case MS_VUID_FORMAT:
                if (uio->uio_resid < sizeof (Firm_event))
                    goto done;
                send_event = 0;
                switch (ms->ms_oldoff1++) {

                    case 0: /* Send x if changed */
                        if (mi->mi_x != 0) {
                            fe.id = vuid_id_addr(ms->ms_vuidaddr) |
                                vuid_id_offset(LOC_X_DELTA);
                            fe.pair_type = FE_PAIR_ABSOLUTE;
                            fe.pair = LOC_X_ABSOLUTE;
                            fe.value = mi->mi_x;
                            send_event = 1;
                        }
                        break;

                    case 1: /* Send y if changed */
                        if (mi->mi_y != 0) {
                            fe.id = vuid_id_addr(ms->ms_vuidaddr) |
                                vuid_id_offset(LOC_Y_DELTA);
                            fe.pair_type = FE_PAIR_ABSOLUTE;
                            fe.pair = LOC_Y_ABSOLUTE;
                            fe.value = -mi->mi_y;
                            send_event = 1;
                        }
                        break;

                    default: /* Send buttons if changed */
                        hwbit = MS_HW_BUT1 >> (ms->ms_oldoff1 - 3);
                        if ((ms->ms_readbuttons & hwbit) !=
                            (mi->mi_buttons & hwbit)) {
                            fe.id = vuid_id_addr(ms->ms_vuidaddr) |
                                vuid_id_offset(
                                    BUT(1) + (ms->ms_oldoff1 - 3));
                            fe.pair_type = FE_PAIR_NONE;
                            fe.pair = 0;
                        }
                }
            }
    }

```

<elided material>

```

        /* Update read buttons and set value */
        if (mi->mi_buttons & hwbit) {
            fe.value = 0;
            ms->ms_readbuttons |= hwbit;
        } else {
            fe.value = 1;
            ms->ms_readbuttons &= ~hwbit;
        }
        send_event = 1;
    }
    /* Increment mouse buffer pointer */
    if (ms->ms_oldoff1 == 5) {
        ms->ms_oldoff++;
        if (ms->ms_oldoff >= b->mb_size)
            ms->ms_oldoff = 0;
        ms->ms_oldoff1 = 0;
    }
    break;
}
if (send_event) {
    fe.time = mi->mi_time;
    ms->ms_vuidcount--;
    /* lower pri to avoid mouse droppings */
    (void) splx(pri);
    error = uiomove(&fe, sizeof(fe), UIO_READ, uio);
    /* spl_ms should return same priority as pri */
    pri = spl_ms();
}
break;
}
}
done:
    /* Flush tty if no more to read */
    if ((ms->ms_oldoff1 == 0) && (ms->ms_oldoff == b->mb_off))
        ttyflush(tp, FREAD);
    /* Release protection AFTER ttyflush or will get out of sync with tty */
    (void) splx(pri);
    return (0);
}

/* Mouse ioctl */
msioctl(tp, cmd, data, flag)
    struct tty *tp;
    int cmd;
    caddr_t data;
    int flag;
{
    register struct msdata *msd = mstptomsd(tp);
    register struct ms_softc *ms;
    int err = 0, num;
    register int buf_off, read_off;

```

```
Void_addr_probe *addr_probe;

if (msd == 0)
    return(EINVAL);
ms = &msd->msd_softc;
switch (cmd) {
case FIONREAD:
    switch (ms->ms_readformat) {
    case MS_3BYTE_FORMAT:
        *(int *)data = ms->ms_samplecount;
        break;

    case MS_VUID_FORMAT:
        *(int *)data = sizeof (Firm_event) * ms->ms_vuidcount;
        break;
    }
    break;

case VUIDSFORMAT:
    if (*(int *)data == ms->ms_readformat)
        break;
    ms->ms_readformat = *(int *)data;
    /*
     * Flush mouse buffer because otherwise ms_*counts
     * get out of sync and some of the offsets can too.
     */
    msflush(msd);
    break;

case VUIDGFORMAT:
    *(int *)data = ms->ms_readformat;
    break;

case VUIDSADDR:
    addr_probe = (Void_addr_probe *)data;
    if (addr_probe->base != VKEY_FIRST) {
        err = ENODEV;
        break;
    }
    ms->ms_vuidaddr = addr_probe->data.next;
    break;

case VUIDGADDR:
    addr_probe = (Void_addr_probe *)data;
    if (addr_probe->base != VKEY_FIRST) {
        err = ENODEV;
        break;
    }
    addr_probe->data.current = ms->ms_vuidaddr;
    break;

case TIOCSETD:
    /*
```



```

    * Don't let the line discipline change once it has been set
    * to a mouse. Changing the ldisc causes msclose to be called
    * even if the ldisc of the tp is the same.
    * We can't let this happen because the window system may have
    * a handle on the mouse buffer.
    * The basic problem is one of having anything depending on
    * the continued existence of ldisc related data.
    * The fix is to have:
    * 1) a way of handing data to the dependent entity, and
    * 2) notifying the dependent entity that the ldisc
    * has been closed.
    */
    break;

```

<elided material>

```

        default:
            err = ttioctl(tp, cmd, data, flag);
        }
    return (err);
}

msflush(msd)
    register struct msdata *msd;
{
    register struct ms_softc *ms = &msd->msd_softc;
    int s = spl_ms();

```

<elided material>

```

        ttyflush(msd->msd_tp, FREAD);
        (void) splx(s);
    }

```

<elided material>

```

/* Called with next byte of mouse data */
/*ARGSUSED*/
msinput(c, tp)
    register char c;
    struct tty *tp;
{
    int s = spl5();

```

<elided material>

```

    /* Place data on circular buffer */

    if (wake)
        /* Place character on tty raw input queue to trigger select */
        ttyinput('\0', msd->msd_tp);
    (void) splx(s);
}

```

```
/* Match tp to msdata */
struct msdata *
mstptomsd(tp)
    struct tty *tp;
{
    register i;

    /* Get next free msdata */
    for (i = 0; i < NMS; ++i)
        if (msdata[i].msd_tp == tp)
            return(&msdata[i]);
    printf("mstptomsd called with unknown tp %X\n", tp);
    return(0);
}
```

B

Programming Notes

Programming Notes	229
B.1. What Is Supported?	229
B.2. Library Loading Order	229
B.3. Shared Libraries vs. Shared Text	229
Shared Libraries	229
B.4. Error Message Decoding	230
B.5. Debugging Hints	230
Disabling Locking	230
B.6. Sufficient User Memory	231
B.7. Coexisting with UNIX	232
Tool Initialization and Process Groups	232
Signals from the Control Terminal	232
Job Control and the C Shell	232



Programming Notes

Here are useful hints for programmers who use SunView.

B.1. What Is Supported?

In each release, there may be some difference between the documentation and the actual product implementation. The documentation describes the supported implementation. In general, the documentation indicates where features are only partially implemented, and in which directions future extensions may be expected. Any necessary modifications to SunView are accompanied by a description of the nature of the changes and appropriate responses to them.

B.2. Library Loading Order

When loading programs, remember to load higher level libraries first, that is, -**lsuntool -lsunwindow -lpixrect**.

B.3. Shared Libraries vs. Shared Text

Starting with release 3.2 of SunOS, the tools in SunView were distributed as two huge *toolmerge* files in order to reduce the working set requirements of SunView. Instead of having many different programs running, each including its own copy of the `libsuntool.a`, `libsunwindow.a` and `libpixrect.a` libraries, several copies of one large program would run at once.

Shared Libraries

With the advent of shared libraries in SunOS release 4.0, toolmerging is unnecessary in most cases. By default, each tool is compiled and linked to use shared libraries, so the code of the SunView 1 libraries is still shared.³³ This also has the benefit of greatly reducing the size of SunView 1 binaries. Another benefit is that programs will automatically benefit from improvements in the SunView 1 libraries in each release, without recompilation.

You may find old programs or Makefiles with references to `-DMERGE` or `-DSTANDALONE` in them.

Should you want to compile a program with static libraries, just supply the `-Bstatic` flag to the link editor or C compiler (read the `ld(1)` man page for details). One reason to use static libraries is that it simplifies part of the debugging process.

³³ The standard C library and many others are shared also.

B.4. Error Message Decoding

The default error reporting scheme described in Section 5.12, *Error Handling*, displays a long hexadecimal number which is the `ioctl` number associated with the error. You can turn this number into a more meaningful operation name by:

- turning the two least significant digits into a decimal number;
- searching `/usr/include/sunwindow/win_ioctl.h` for occurrences of this number; and
- noting the `ioctl` operation associated with this number.

This can provide a quick hint as to what is being complained about without resorting to a debugger.

B.5. Debugging Hints

When debugging non-terminal oriented programs in the window system, there are some things that you should know to make things easier.

As discussed mentioned in passing a process receives a `SIGWINCH` whenever one of its windows changes state. In particular, as soon as a frame is shown, the kernel sends it a `SIGWINCH`. When running as the child of a debugger, the `SIGWINCH` is sent to the parent debugger instead of to the tool. By default, `dbx` simply propagates the `SIGWINCH` to the tool, while `adb` traps, leaving the tool suspended until the user continues from `adb`. This behavior is not peculiar to `SIGWINCH`: `adb` traps all signals by default, while `dbx` has an initial list of signals (including `SIGWINCH`) that are passed on to the child process. You can instruct `adb` to pass `SIGWINCH` on to the child process by typing `1c:1` Return. `1c` is the hex number for 28, which is `SIGWINCH`'s number. Re-enable signal breaking by typing `1c:t` Return. You can instruct `dbx` to trap on a signal by using the `catch` command.

For further details, see the entries for the individual debuggers in the *User's Manual for the Sun Workstation*. In addition, `ptrace(2)` describes the fine points of how kernel signal delivery is modified while a program is being debugged.

The two debuggers differ also in their abilities to interrupt programs built using tool windows. `dbx` knows how to interrupt these programs, but `adb` doesn't. See *Signals from the Control Terminal* below for an explanation.

Disabling Locking

Another situation specific to the window system is that various forms of locking are done that can get in the way of smooth debugging while working at low levels of the system. There are variables in the `sunwindow` library that disable the actual locking. These variables can be turned on from a debugger:

Table B-1 *Variables for Disabling Locking*

<i>Variable/Action</i>	
<code>int pixwindebug</code>	When not zero this causes the immediate release of the display lock after locking so that the debugger is not continually getting hung by being blocked on writes to screen. Display garbage can result because of this action.
<code>int win_lockdatadebug</code>	When not zero, the data lock is never actually locked, preventing the debugger from being continually hung during block writes to the screen. Unpredictable things may result because of this action that can't properly be described in this context.
<code>int win_grabiodebug</code>	When not zero will not actually acquire exclusive I/O access rights so that the debugger wouldn't get hung by being blocked on writes to the screen and not be able to receive input. The debugged process will only be able to do normal display locking and be able to get input only in the normal way.
<code>int fullscreendebug</code>	Like <code>win_grabiodebug</code> but applies to the fullscreen access package.

Change these variables only during debugging. You can set them any time after `main` has been called.

B.6. Sufficient User Memory

To use the SunView environment comfortably requires adequate user memory for SunView and SunOS, Sun's operating system. To achieve the best performance, **you must** reconfigure your own kernel, deleting unused device drivers and possibly reducing some tuning parameters. The procedure is documented in the manual *Installing the SunOS*. You will be able to reclaim a significant amount of usable main memory.

B.7. Coexisting with UNIX

This section discusses how a SunView tool interacts with traditional UNIX features in the areas of process groups, signal handling, job control and terminal emulation. If you are not familiar with these concepts, read the appropriate portions (*Process Groups*, *Signals*) of the *System Services Overview*, and the `signal(3)` and `tty(4)` entries in the *SunOS Reference Manual*.

This discussion explicitly notes those places where the shells and debuggers interact differently with a tool.

Tool Initialization and Process Groups

System calls made by the library code in a tool affect the signals that will be sent to the tool. A tool acts like any program when first started: it inherits the process group and control terminal group from its parent process. However, when a frame is created, the tool changes its process group to its own process number. The following sections describe the effects of this change.

Signals from the Control Terminal

When the C shell (see `csh(1)`) starts a program, it changes the process group of the child to the child's process number. In addition, if that program is started in the foreground, the C shell also modifies the process group of the control terminal to match the child's new process group. Thus, if the tool was started from the C shell, the process group modification done by `window_create()` has no effect.

The Bourne Shell (see `sh(1)`) and the standard debuggers do not modify their child's process and control terminal groups. Furthermore, both the Bourne Shell and `adb(1)` are ill-prepared for the child to perform such modification. They do not propagate signals such as `SIGINT` to the child because they assume that the child is in the same control terminal group as they are. The bottom-line is that when a tool is executed by such a parent, typing interrupt characters at the parent process does not affect the child, and vice versa. For example, if the user types an interrupt character at `adb` while it is debugging a tool, the tool is not interrupted. Although `dbx(1)` does not modify its child's process group, it is prepared for the child to do so.

Job Control and the C Shell

The terminal driver and C shell job control interact differently with tools. First, let us examine what happens to programs using the graphics subwindow library package³⁴ When the user types an interrupt character on the control terminal, a signal is sent to the executing program. When the signal is a `SIGTSTP`, the `gfxsw` library code sees this signal and releases any SunView locks that it might have and removes the graphics from the screen before it actually suspends the program. If the program is later continued, the graphics are restored to the screen.

However, when the user types the C shell's `stop` command to interrupt the executing program, the C shell sends a `SIGSTOP` to the program and the `gfxsw` library code has no chance to clean up. This causes problems when the code has acquired any of the SunView locks, as there is no opportunity to release them. Depending on the lock timeouts, the kernel will eventually break the locks, but

³⁴ The `gfxsubwindow` is an out-dated package used only as an example.

until then, the entire screen is unavailable to other programs and the user. To avoid this problem, the user should send the C shell `kill` command with the `-TSTP` option instead of using `stop`.

The situation for tools parallels that of the `gfxsw` code. Thus a tool that wants to interact nicely with job control must receive the signals related to job control `>L SIGINT`, `(SIGQUIT`, and `SIGTSTP)` and release any locks it has acquired. If the tool is later continued, the tool must receive a `SIGCONT` so that it can reacquire the locks before resuming the window operations it was executing. The tool will still be susceptible to the same problems as the `gfxsw` code when it is sent a `SIGSTOP`.

A final note: the user often relies on job control without realizing it; the expectation is that typing interrupt characters will halt a program. Of course, even programs that do not use SunView facilities, such as a program that opens the terminal in "raw" mode, have to provide a way to terminate the program. A program using the `gfxsw` package that reads any input can provide limited job control by calling `gfxsw_inputinterrupts`.

Index

A

adb(1), 232
the Agent, 21, 13
 notification of tiles, 23
 posting notification to tiles, 24
 removing a tile from, 26
 SunView 1 model, 14
architecture, 7
ASCII_DEVID, 54

B

bell, 37
blanket window, 41
bool, 197
Bourne Shell, 232

C

C shell, 232
 job control, 232
caret, 59, 96
client handle, used by win_register(), 21
clipping, 14
close(2), 91
color
 screen foreground and background colors, 48
colormap
 segment, 16
 segments, 15
 shared, 16
coord type for Rects, 197
Copy function-key processing, 104
csh(1), 232
cursor, 15
Cut function-key processing, 104

D

dbx, 230, 232
debugging, 230
 disabling locking, 230
 fullscreendebug, 230
 pixwindebug, 230
 win_grabidebug, 230
 win_lockdatadebug, 230
defaults, 145
 creating .d files, 149
 default value, 146

defaults, *continued*

 directory, 146
 distinguished names, 148
 enumerated option, 148
 error handling, 156
 Error_Action, 150, 156
 example, 149
 example program, 160
 master and private defaults databases, 146
 Maximum_Errors, 157
 option names, 147
 option values, 148
 private options, 146
 Private_Directory, 146
 Private_only, 147
 Test_Mode, 157
 Testmode, 146
defaults database, 145
defaults functions
 defaults_exists(), 157
 defaults_get(), 150
 defaults_get_boolean(), 151, 157
 defaults_get_character(), 151, 157
 defaults_get_child(), 157
 defaults_get_default(), 158
 defaults_get_enum(), 152, 158
 defaults_get_integer(), 150, 158
 defaults_get_integer_check(), 151, 158
 defaults_get_sibling(), 158
 defaults_get_string(), 150, 158
 defaults_reread(), 158
 defaults_set_character(), 158
 defaults_set_enumeration(), 159
 defaults_set_integer(), 159
 defaults_set_string, 159
 defaults_special_mode(), 159
 defaults_write_all(), 159
 defaults_write_changed(), 159
 defaults_write_differences(), 159
<sunwindow/defaults.h>, 145
DEFAULTS_UNDEFINED, 149
desktop, 47, 11, 15, — *also see* screen
 accessing the root fd, 50
 foreground and background colors, 48
 frame buffer, 48
 keyboard, 48
 locking, 16
 mouse, 48

desktop, *continued*
 root window, 48
 screen, 48
 DESTROY_CHECKING, 80, 85
 DESTROY_PROCESS_DEATH, 80
 display
 locking, 16
 documentation
 outline of this document, 7
 pixrect vs. application vs. system manuals, 3
 dup(2), 91

E

enumerated values
 retrieving, 152
 environment
 SunView application usage, 42
 window usage, 41
 error
 WIN ioctl errors, 43, 230
 Error_action, 150
 event
 client, 68, 81
 client event func(), 69
 delivery, 68, 77
 dispatching, 67, 68, 82, 86
 handlers, 67
 interposition, 69
 ordering, 67, 81
 posting, 68, 69, 77
 posting client events, 77
 posting destroy, 80
 retrieving event handler, 70
 event codes
 SCROLL_ENTER, 206
 SCROLL_EXIT, 206
 SCROLL_REQUEST, 207
 event_func(), 24, 69
 EWOULDBLOCK, 39
 example programs
 defaults, 160
 filer_default, 160
 get_selection, 125
 seln_demo, 128
 exception_func(), 70

F

FALSE, 197
 FASYNC, 39
 FBIONREAD, 40
 fcntl(), 39
 fd_set, 82
 FE_PAIR_ABSOLUTE, 215
 FE_PAIR_DELTA, 215
 FE_PAIR_NONE, 215
 FE_PAIR_SET, 215
 file descriptor
 window, 13, 30
 Find function-key processing, 104
 FIOASYNC, 39, 91
 FIONBIO, 39, 91

Firm_event, 59, 215
 flash, 37
 FNDELAY, 39
 focus, 58
 frame buffer, 48
 fullscreen, 181
 coordinate space, 181
 grab I/O, 182
 initializing, 182
 pixel caching, 183
 pixwin operations, 184
 saving and restoring image, 183
 struct fullscreen, 181
 surface preparation, 182
 use menu and alert packages instead, 181
 use window_loop() instead, 181
 window boundary violation, 183
 fullscreen functions
 fullscreen_destroy(), 182
 fullscreen_init(), 182
 fullscreen_pw_copy(), 185
 fullscreen_pw_vector(), 184
 fullscreen_pw_write(), 184
 pw_preparesurface(), 183
 pw_restore_pixels(), 184
 pw_save_pixels(), 183
 fullscreendebug, 230
 function key processing, 104

G

geometry of rectangles, *see* rect
 get_selection example program, 125
 gfxsw, 233

H

header files
 <sundev/vuid_event.h>, 55, 215
 <suntool/scrollbar.h>, 205
 <suntool/selection_attributes.h>, 110
 <suntool/selection_svc.h>, 110
 <sunwindow/attr.h>, 110
 <sunwindow/defaults.h>, 145
 <sunwindow/rect.h>, 197
 <sunwindow/rectlist.h>, 197
 <sunwindow/win_enum.h>, 35
 <sunwindow/win_input.h>, 55
 <sunwindow/window_hs.h>, 29

I

icon, 174
 dynamic loading, 174
 file format, 174
 template, 174
 icon_init_from_pr(), 174
 icon_load(), 174
 icon_load_mpr(), 174
 icon_open_header(), 175
 IM_ASCII, 38
 IM_INTRANSIT, 38
 IM_META, 38
 IM_NEGASCII, 38

IM_NEGEVENT, 38
 IM_NEGMETA, 38
 imaging
 fixup, 173
 input, 37, *see (mostly)* workstation and vuid
 SIGIO, 39
 ascii, 38
 asynchronous, 39
 blocking, 39
 bytes pending, 40
 caret, 59
 changing interrupt user actions, 63
 current event, 60
 current event lock, 60
 current event lock broken, 60
 designee, 39
 events pending, 40
 flow of control, 39
 keyboard mask, 39
 masks, 37
 meta, 38
 negative events, 38
 non-blocking, 39
 pick mask, 39
 reading, 39
 redirection, 39
 releasing the current event lock, 60
 seizing all, 182
 state, 55
 synchronization, 60, 61
 synchronous, 39
 unencode, 55
 input device
 control, 56
 enumeration, 58
 query, 57
 removal, 57
 setting and initialization, 57
 input focus, 58, 59
 getting the caret event, 59
 keyboard, 58
 restoring the caret, 59
 setting the caret event, 59
 input_imnull(), 38
 input_readevent, 39
 inputmask, 37
 ioctl(2), 91
 ITIMER_REAL, 81
 ITIMER_VIRTUAL, 81

J

job control, 232

K

KBD_REQUEST, 58
 kernel tuning, 61
 win_disable_shared_locking, 62
 winlistcharsmax, 62
 ws_check_lock, 62
 ws_check_time, 62
 ws_fast_poll_duration, 62
 ws_fast_timeout, 61

kernel tuning, *continued*
 ws_loc_still, 62
 ws_lock_limit, 62
 ws_set_favor, 62
 ws_slow_timeout, 62
 ws_vq_node_bytes, 61
 keyboard, 56
 focus, 58
 unencoded input, 55
 KIOCTRANS, 55

L

LOC_RGNENTER, 24
 LOC_RGNEXIT, 24

M

menu, *see* fullscreen
 mouse, 56
 sample vuid driver, 218

N

Notifier, 67, 13
 advanced usage, 67
 client, *see* client
 client event handlers, 68
 client events, 68
 copy_func() calling sequence, 79
 destroy event delivery time, 80
 error codes, 88
 event delivery time, 77
 exception event handlers, 70
 interaction with various system calls, 91
 interposition, 72
 miscellaneous issues, 91
 notification, 68
 output event handlers, 69
 output events, 69
 posting destroy events, 80
 posting events, 77
 posting events with an argument, 78
 prioritization, 81
 registering an interposer, 72
 release_func() calling sequence, 79
 restrictions, 67, 90
 safe destruction, 80
 storage management during event posting, 78
 Notifier functions
 notify_client(), 86
 notify_die(), 85
 notify_event(), 82
 notify_exception(), 83
 notify_get_client_func(), 70
 notify_get_destroy_func(), 71
 notify_get_event_func(), 70
 notify_get_exception_func(), 70, 71
 notify_get_input_func(), 70
 notify_get_itimer_func(), 71
 notify_get_output_func(), 70
 notify_get_prioritizer_func(), 83
 notify_get_scheduler_func(), 87
 notify_get_signal_func(), 71
 notify_get_wait3_func(), 71
 notify_input(), 82

Notifier functions, *continued*

notify_interpose_destroy_func(), 73
 notify_interpose_event_func(), 72
 notify_interpose_exception_func(), 73
 notify_interpose_input_func(), 72
 notify_interpose_itimer_func(), 73
 notify_interpose_output_func(), 73
 notify_interpose_signal_func(), 73
 notify_interpose_wait3_func(), 73
 notify_itimer(), 83
 notify_next_destroy_func(), 74
 notify_next_event_func(), 72, 74
 notify_next_exception_func(), 74
 notify_next_input_func(), 74
 notify_next_itimer_func(), 74
 notify_next_output_func(), 74
 notify_next_signal_func(), 74
 notify_next_wait3_func(), 74
 notify_output(), 82
 notify_perror(), 89
 notify_post_destroy(), 80, 85
 notify_post_event(), 69, 77
 notify_post_event_and_arg(), 78, 79
 notify_remove(), 87
 notify_remove_destroy_func(), 75
 notify_remove_event_func(), 75
 notify_remove_exception_func(), 75
 notify_remove_input_func(), 75
 notify_remove_itimer_func(), 75
 notify_remove_output_func(), 75
 notify_remove_signal_func(), 75
 notify_remove_wait3_func(), 75
 notify_set_event_func(), 68, 72
 notify_set_exception_func(), 70
 notify_set_output_func(), 69
 notify_set_prioritizer_func(), 81
 notify_set_scheduler_func(), 86
 notify_signal(), 83
 notify_start(), 85
 notify_stop(), 85
 notify_veto_destroy(), 85
 notify_wait3(), 83
 Notify_arg, 69, 79
 NOTIFY_BAD_FD, 70
 NOTIFY_BAD_ITIMER, 88
 NOTIFY_BAD_SIGNAL, 88
 NOTIFY_BADF, 88
 NOTIFY_CLIENT_NULL, 86
 Notify_copy, 79
 NOTIFY_COPY_NULL, 79
 NOTIFY_DESTROY_VETOED, 80, 85, 88
 NOTIFY_DONE, 69, 82
 notify_errno, 73, 88
 Notify_error, 88
 Notify_event, 77
 Notify_event_type, 68
 NOTIFY_FUNC_LIMIT, 73, 89
 NOTIFY_FUNC_NULL, 70, 84
 NOTIFY_IGNORED, 69, 77, 82
 NOTIFY_IMMEDIATE, 68, 77
 NOTIFY_INTERNAL_ERROR, 88
 NOTIFY_INVALID, 80, 89

NOTIFY_NO_CONDITION, 70, 73, 77, 83, 88
 NOTIFY_NOMEM, 88
 NOTIFY_NOT_STARTED, 85, 88
 NOTIFY_OK, 73, 88
 Notify_release, 79
 NOTIFY_RELEASE_NULL, 79
 NOTIFY_SAFE, 68, 77
 NOTIFY_SRCH, 88
 NOTIFY_UNKNOWN_CLIENT, 70, 73, 77, 83, 88

O

output_func(), 69

P

PANEL_DEVID, 54
 Paste function-key processing, 104
 pixwin, 13
 closing, 37
 colormap, 16
 colormap segment, 15
 creation, 36
 damage, 176
 damage report, 177
 destruction, 37
 flashing, 37
 locking, 16
 offset control, 178
 opening, 36
 region, 21
 repair of retained pixwin, 177
 repairing damage, 176
 retained, 177
 signals, 176
 SIGWINCH, 176
 surface preparation, 183
 pixwin functions and macros
 pw_close(), 37
 pw_damaged(), 176
 pw_donedamaged(), 177
 pw_exposed(), 173
 pw_get_x_offset(), 178
 pw_get_y_offset(), 178
 pw_open(), 21, 36
 pw_preparesurface(), 183
 pw_region(), 21
 pw_repairretained(), 177
 pw_restore_pixels(), 184
 pw_restrict_clipping(), 173
 pw_save_pixels(), 183
 pw_set_region_rect(), 24
 pw_set_x_offset(), 178
 pw_set_xy_offset(), 178
 pw_set_y_offset(), 178
 pixwindebug, 230
 prioritizer_func(), 81
 prompt, *see* fullscreen
 PW_FIXED_IMAGE, 21, 22
 PW_INPUT_DEFAULT, 21, 22
 PW_NO_LOC_ADJUST, 21, 22
 Pw_pixel_cache(), 183
 PW_PIXEL_CACHE_NULL, 183

PW_REPAINT_ALL, 21, 22
 PW_RETAIN, 21, 22

R

“raw” mode, 233
 readv(2), 91
 rect, 197, 31
 coord type, 197
 Rectlist, 197
 rect and rectlist functions and macros
 rect_bottom(), 197
 rect_bounding(), 198
 rect_clipvector(), 199
 rect_construct(), 198
 rect_equal(), 198
 rect_includespoint(), 198
 rect_includesrect(), 198
 rect_intersection(), 198
 rect_intersectsrect(), 198
 rect_isnull(), 198
 rect_marginadjust(), 198
 rect_order(), 199
 rect_passtochild(), 198
 rect_passtoparent(), 198
 rect_right(), 197
 rl_boundintersectsrect(), 201
 rl_coalesce(), 202
 rl_coordoffset(), 200
 rl_copy(), 202
 rl_difference(), 202
 rl_empty(), 201
 rl_equal(), 201
 rl_equalrect(), 201
 rl_free(), 202
 rl_includespoint(), 201
 rl_initwithrect(), 202
 rl_intersection(), 202
 rl_normalize(), 202
 rl_null(), 200
 rl_passtochild(), 200
 rl_passtoparent(), 200
 rl_rectdifference(), 202
 rl_rectintersection(), 202
 rl_rectoffset(), 200
 rl_rectunion(), 202
 rl_sort(), 202
 rl_union(), 202
 Rect struct, 197
 rect_null, 198
 Rectlist, 199
 rectnode, 200
 RECTS_BOTTOMTOTOP, 199
 RECTS_LEFTTORIGHT, 199
 RECTS_RIGHTTOLEFT, 199
 RECTS_SORTS, 199
 RECTS_TOPTOBOTTOM, 199
 RECTS_UNSORTED, 199
 region
 use for tiles, 21
 rlimit(2), 91
 root window
 accessing the root fd, 50

S

scheduler_func(), 86
 SCR_EAST, 50
 SCR_NAMESIZE, 48, 57
 SCR_NORTH, 50
 SCR_POSITIONS, 50
 SCR_SOUTH, 50
 SCR_SWITCHBKGRDFRGRD, 48
 SCR_WEST, 50
 screen, 11, 47, 48, 56
 adjacent, 50
 creating, 48
 destruction, 49
 fullscreen access, *see* fullscreen
 mouse, 56
 multiple, 50
 positions, 50
 querying, 49
 standard command-line argument parsing, 49
 SCROLL_DEVID, 54
 SCROLL_ENTER, 206
 SCROLL_EXIT, 206
 SCROLL_REQUEST, 207
 scrollbar, 205
 line-by-line scrolling, 211
 scrolling, 208
 updating, 206
 scrollbar attributes
 SCROLL_LAST_VIEW_START, 210
 SCROLL_LINE_HEIGHT, 211
 SCROLL_MARGIN, 209
 SCROLL_NORMALIZE, 209
 SCROLL_NOTIFY_CLIENT, 205
 SCROLL_OBJECT, 206, 210
 SCROLL_OBJECT_LENGTH, 206, 210
 SCROLL_REQUEST_MOTION, 210
 SCROLL_REQUEST_OFFSET, 210
 SCROLL_VIEW_LENGTH, 206, 210
 SCROLL_VIEW_START, 209, 210
 <suntool/scrollbar.h>, 205
 selection callback procedures, 104, 97, 102
 function_proc, 105, 113
 reply_proc, 113
 reply_proc(), 106
 selection client debugging
 adjusting RPC timeouts, 109
 dumping selection data, 109
 running a test service, 109
 service displays, 109
 tracing request attributes, 123
 selection library data types
 Seln_function, 110
 Seln_function_buffer, 105, 111
 Seln_holder, 110
 Seln_holders_all, 111
 Seln_rank, 110
 Seln_replier_data, 107, 111
 Seln_request, 102, 111
 Seln_requester, 102, 111
 Seln_response, 105, 110
 Seln_result, 110
 Seln_state, 110

selection library procedures

- `seln_acquire()`, 104, 112
- `seln_ask()`, 103, 112
- `seln_clear_functions()`, 112
- `seln_create()`, 96, 98, 112
- `seln_debug()`, 113
- `seln_destroy()`, 97, 113
- `seln_done()`, 113
- `seln_dump_function_buffer()`, 113
- `seln_dump_function_key()`, 114
- `seln_dump_holder()`, 114
- `seln_dump_rank()`, 114
- `seln_dump_response()`, 114
- `seln_dump_result()`, 114
- `seln_dump_service()`, 114
- `seln_dump_state()`, 115
- `seln_figure_response()`, 105, 115
- `seln_functions_state()`, 102, 115
- `seln_get_function_state()`, 102, 115
- `seln_hold_file()`, 109, 115
- `seln_holder_same_client()`, 116
- `seln_holder_same_process()`, 116
- `seln_inform()`, 116
- `seln_init_request()`, 103, 117
- `seln_inquire()`, 117, 126
- `seln_inquire_all()`, 117
- `seln_query()`, 103, 118, 126
- `SELN_REPORT`, 118
- `seln_report_event()`, 102, 116
- `seln_report_event()`, 118
- `seln_request()`, 103, 119
- `seln_same_holder()`, 119
- `seln_secondary_exists()`, 106, 119
- `seln_secondary_made()`, 106, 119
- `seln_use_test_service()`, 109, 119
- `seln_use_timeout()`, 109, 120
- `seln_yield()`, 104
- `seln_yield_all()`, 120

selection request, 102, 95

- buffer, 102
- buffer size, 111
- for non-held selection, 108
- initiated by function-key, 104
- long replies, 104, 108
- read procedure, 103
- replier context, 107
- replying, 106
- request attribute definitions, 103
- requester context, 104
- sample program, 104
- unrecognized requests, 104, 108

selection request attributes, 121 *thru* 124

- `SELN_REQ_BYTESIZE`, 122
- `SELN_REQ_COMMIT_PENDING_DELETE`, 106, 108, 123
- `SELN_REQ_CONTENTS_ASCII`, 103, 122
- `SELN_REQ_CONTENTS_PIECES`, 122
- `SELN_REQ_DELETE`, 123
- `SELN_REQ_END_REQUEST`, 107, 124
- `SELN_REQ_FAILED`, 124
- `SELN_REQ_FAKE_LEVEL`, 123
- `SELN_REQ_FILE_NAME`, 122
- `SELN_REQ_FIRST`, 122
- `SELN_REQ_FIRST_UNIT`, 122
- `SELN_REQ_LAST`, 122

selection request attributes, *continued*

- `SELN_REQ_LAST_UNIT`, 122
- `SELN_REQ_LEVEL`, 122
- `SELN_REQ_RESTORE`, 123
- `SELN_REQ_SET_LEVEL`, 123
- `SELN_REQ_UNKNOWN`, 124
- `SELN_REQ_UNRECOGNIZED`, 104
- `SELN_REQ_YIELD`, 106, 108, 123

selection response

- `SELN_DELETE`, 106
- `SELN_FIND`, 106
- `SELN_IGNORE`, 105
- `SELN_REQUEST`, 106
- `SELN_SHELVES`, 106

Selection Service, 95, 13

Selection Service

- acquiring selections, 104
- adjusting RPC timeouts, 109
- callback procedures, 97, 102, 104, 112
- caret, 96
- client, 96
- clipboard, 96
- common request attributes, 121
- concepts, 96
- consume-reply procedure, 103
- data definitions, 110
- debugging clients, *see* selection client debugging
- enumerated types, 110
- function key notifications and processing, 104
- function key transitions, 101, 105
- getting the selection's contents, 102
- header files, 110
- library, 95
- overview, 96
- Primary, 96
- procedure declarations, 112
- protocol example, 97
- releasing selections, 104
- replying to requests, 106
- request, *see* selection request
- request buffers, 102
- required header files, 110
- running a test service, 109
- sample program, 104
- sample program *get_selection*, 125
- sample program *seln_demo*, 128
- sample programs, 125
- Secondary, 96
- selection holder, 96
- selection rank, 96
- `selection_svc(1)` program, 97
- sending requests to the selection holder, 102
- server process, 95
- shelf, 96
- status display & tracing, 109
- the selection itself, 96
- timeouts, 109
- tracing request attributes, 123
- `SELN_CARET`, 110
- `SELN_CONTINUED`, 110
- `SELN_DELETE`, 110
- seln_demo* example program, 128
- `SELN_DIDNT_HAVE`, 110

SELN_EXISTS, 110
 SELN_FAILED, 110
 SELN_FILE, 110
 SELN_FIND, 110
 SELN_FN_AGAIN, 110
 SELN_FN_ERROR, 110
 SELN_FN_FIND, 110
 SELN_FN_FRONT, 110
 SELN_FN_GET, 110
 SELN_FN_OPEN, 110
 SELN_FN_PROPS, 110
 SELN_FN_PUT, 110
 SELN_FN_STOP, 110
 SELN_FN_UNDO, 110
 SELN_IGNORE, 110
 SELN_LEVEL_ALL, 124
 SELN_LEVEL_FIRST, 124
 SELN_LEVEL_LINE, 124
 SELN_LEVEL_NEXT, 124
 SELN_LEVEL_PREVIOUS, 124
 SELN_NON_EXIST, 110
 SELN_NONE, 110
 SELN_PRIMARY, 110
 seln_*, *see* selection library procedures
 SELN_REQ_*, *see* selection request attributes
 SELN_REQUEST, 110
 SELN_SECONDARY, 110
 SELN_SHELF, 110
 SELN_SHELVE, 110
 SELN_SUCCESS, 110
 SELN_TRACE_ACQUIRE, 123
 SELN_TRACE_DONE, 123
 SELN_TRACE_HOLD_FILE, 123
 SELN_TRACE_INFORM, 123
 SELN_TRACE_INQUIRE, 123
 SELN_TRACE_STOP, 123
 SELN_TRACE_YIELD, 123
 Seln_*, *see* selection library data types
 SELN_UNKNOWN, 110
 SELN_UNRECOGNIZED, 110
 SELN_UNSPECIFIED, 110
 SELN_WRONG_RAN, 110
 SELN_WRONG_RANK, 110
 setjmp(2), 91
 setpriority(2), 91
 setquota(2), 91
 SIG_BIT, 81
 SIGALRM, 83
 sigblock(2), 91
 SIGCHLD, 83
 SIGIO, 39
 SIGKILL, 49
 sigmask(2), 91
 signal(3), 232
 signal handling, 232
 signal(2), 90
 signal(3), 91
 sigpause(2), 91

sigstack(2), 91
 SIGTERM, 49, 80
 SIGTSTP, 232
 SIGURG, 70
 sigvec(2), 90, 91
 SIGVTALRM, 83
 SIGWINCH, 42, 176, 230
 SIGXCPU, 34
 singlecolor, 47
 SunView
 abstractions/objects, 11
 architecture, 7, 12
 changes from 2.0, 3
 compatibility with future releases, 3
 environment usage, 42
 introduction, 3
 no more tool merging, 229
 programming notes, 229
 shared libraries, 229
 system model, 11
 what is supported, 229
 SUNVIEW_DEVID, 54
 system calls not to be used under SunView, 91

T

terminal emulation, 232
 tile, 21, 11
 dynamically changing flags, 23
 extracting data, 23
 laying out, 22
 notification from the Agent, 23
 notifying tiles through the Agent, 24
 overlap, 14
 registering with the Agent, 21
 removing from the Agent, 26
 SunView 1 model, 14
 tool
 iconic flag, 40
 parent, 42
 TOP_DEVID, 54
 TR_UNTRANS_EVENT, 56
 TRUE, 197
 tty(4), 232

U

umask(2), 91
 UNIX and SunView, 232
 UNIX system calls and SunView
 close(2), 91
 dup(2), 91
 ioctl(2), 91
 open(2), 29
 readv(2), 91
 rlimit(2), 91
 select(2), 176
 setitimer(2), 91
 setjmp(2), 91
 setpriority(2), 91
 setquota(2), 91
 sigblock(2), 91
 sigmask(2), 91

UNIX system calls and SunView, *continued*

- signal(2), 90
- signal(3), 91
- sigpause(2), 91
- sigstack(2), 91
- sigvec(2), 90, 91
- write(2), 91

V

Virtual User Input Device, *see* void

void, 53

- adding a new segment, 55
- choosing events, 217
- device controls, 217
- example code, 218
- firm events, 215
- input device control — *see* input device, 56
- no missing keys, 55
- pair, 216
- result values, 55
- sample device driver, 218
- segments, 54
- state, 55
- station codes, 54
- writing a void driver, 215

Void_addr_probe, 218

<sundev/void_event.h>, 55, 215, 218

VUID_FIRM_EVENT, 217

VUID_NATIVE, 217

VUID_SEG_SIZE, 54

VUIDGADDR, 218

VUIDGFORMAT, 217

VUIDSADDR, 218

VUIDSFORMAT, 217

W

we_getgfxwindow(), 41

we_getparentwindow(), 43

we_setgfxwindow(), 41

we_setparentwindow(), 42

when_hint, 77

WIN ioctl number error, 43, 230

win_bell(), 37

win_computeclipping(), 192

win_copy_event(), 25

<sunwindow/win_enum.h>, 35

win_enum_input_device(), 58

win_enumerate_children(), 35

win_enumerate_subtree(), 35

win_error(), 43

win_errorhandler(), 43

win_fdtoname(), 30

win_fdtonumber(), 31

win_findintersect(), 41

win_free_event(), 25

win_get_designee(), 39

win_get_event_timeout(), 61

win_get_fd(), 23

win_get_flags(), 23

win_get_focus_event(), 59

win_get_kbd_focus(), 58

win_get_kbd_mask(), 39

win_get_pick_mask(), 39

win_get_pixwin(), 23

win_get_swallow_event(), 59

win_get_tree_layer(), 36

win_get_vuid_value(), 55

win_getheight(), 31

win_getinputcodebit(), 38

win_getlink(), 33

win_getnewwindow(), 29

win_getowner(), 42

win_getrect(), 31

win_getsavedrect(), 32

win_getscreenpositions(), 50

win_getsize(), 31

win_getuserflags(), 40

win_getwidth(), 31

win_grabio(), 182

win_grabiodebug, 230

win_initscreenfromargv(), 49

<sunwindow/win_input.h>, 55

win_insert(), 33

win_insertblanket(), 41

win_is_input_device(), 57

win_isblanket(), 42

win_lockdata(), 34, 190

win_lockdatadebug, 230

WIN_NAMESIZE, 30

win_nametonenumber(), 30

win_nextfree(), 30

WIN_NULLLINK, 30, 32

win_numbertoname(), 30

win_partialrepair(), 192

win_post_event(), 25

win_post_event_arg(), 25

win_post_id(), 24

win_post_id_and_arg(), 25

win_refuse_kbd_focus, 58

win_register(), 21

win_release_event_lock(), 60

win_releaseio(), 182

win_remove(), 34

win_remove_input_device(), 57

win_removeblanket(), 42

win_screendestroy(), 49

win_screenget(), 49

win_screennew(), 48

win_set_designee(), 39

win_set_event_timeout(), 61

win_set_flags(), 23

win_set_focus_event(), 59

win_set_input_device(), 57

win_set_kbd_focus(), 58

win_set_kbd_mask(), 39

win_set_pick_mask(), 39

win_set_swallow_event(), 59

win_setinputcodebit(), 38

- win_setkbd(), 56
- win_setlink(), 33
- win_setmouseposition(), 40
- win_setms(), 56
- win_setowner(), 42
- win_setrect(), 31
- win_setsavedrect(), 32
- win_setscreenpositions(), 50
- win_setuserflag(), 40
- win_setuserflags(), 40
- win_unlockdata(), 34, 190
- win_unregister(), 26
- win_unsetinputcodebit(), 38
- window, 29, 11
 - activation, 33
 - as device, 29
 - as screen, 47
 - blanket, 41
 - blanket flag, 40
 - clipping, 14
 - creation, 29
 - cursor tracking, 15
 - data, 14
 - data lock, 34
 - database locking, 16
 - decoding error messages, 43
 - destruction, 29
 - device, 13
 - display tree, 14, 32
 - enumerating offspring, 35
 - enumerating the window tree, 36
 - enumeration, 35
 - environment usage, 41
 - errors, 43
 - geometry, 31
 - hierarchy, *see* window — display tree
 - iconic flag, 40
 - identifier conversion, 30
 - input, 37
 - input events, 15
 - locate window, 41
 - mouse position, 40
 - naive programs, 41
 - name, 30
 - new, 29
 - next available, 30
 - null, 30
 - number, 30
 - owner, 29, 42
 - parent, 31
 - position, 14, 31
 - querying size, 31
 - reference, 29
 - referencing, 30
 - saved rect, 32
 - screen information, 49
 - SIGWINCH, 42
 - user data, 40
 - user flags, 40
 - window driver, 13
- window attributes
 - WIN_FD, 29
- window device layer, 7
- window display tree
 - SIGXCPU deadlock resolution, 34
 - batched updates, 34
 - insertion, 33
 - links, 32
 - removal, 34
- window functions and macros
 - window_create, 232
 - window_main_loop(), 85
- window management, 189
 - minimal repaint, 192
- WINDOW_GFX, 41
- Window_handle, 35
- WINDOW_PARENT, 42
- windowfd, 30
- WL_BOTTOMCHILD, 32
- WL_COVERED, 32
- WL_COVERING, 32
- WL_ENCLOSING, 32
- WL_OLDER_SIB, 32
- WL_OLDESTCHILD, 32
- WL_PARENT, 32
- WL_TOPCHILD, 32
- WL_YOUNGERSIB, 32
- WL_YOUNGESTCHILD, 32
- wmgr_bottom(), 189
- wmgr_changelevel(), 191
- wmgr_changerect(), 189
- wmgr_close(), 189
- wmgr_completechangerect(), 190
- wmgr_confirm(), 190
- wmgr_forktool(), 190
- wmgr_getrectalloc(), 192
- WMGR_ICONIC, 191
- wmgr_iswindowopen(), 191
- wmgr_move(), 189
- wmgr_open(), 189
- wmgr_refreshwindow(), 189
- wmgr_setrectalloc(), 192
- wmgr_stretch(), 189
- wmgr_top(), 189
- wmgr_winandchildrenexposed(), 191
- workstation, 53, 11
- WORKSTATION_DEVID, 55, 216
- write(2), 91
- ws_* variables, *see* kernel tuning
- ws_usr_async, 63
- WUF_WMGR1, 191

Notes

Notes

Notes

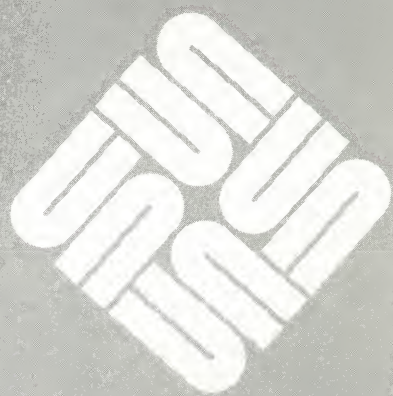
Notes

Notes



Introduction

Introduction	3
What is SunView?	3
History	4
Release 3.0	4
Release 3.2	4
Release 3.4	4
Release 3.5	4
Release 4.0	4
On-Line Help	5
Code No Longer Supported	5



Introduction

What is SunView?

SunView (Sun Visual/Integrated Environment for Workstations) is a user-interface toolkit to support interactive, graphics-based applications running within windows. It consists of two major areas of functionality: building blocks for output, and a run-time system for managing input. The building blocks include four types of windows:

- *canvases* on which programs can draw,
- *text subwindows* with built in editing capabilities,
- *panels* containing items such as buttons, choice items, and analog sliders,
- *tty subwindows* in which programs can be run.

Canvases, text subwindows, and panels can be scrolled.

These windows are arranged as *subwindows* within *frames*, which are themselves windows. Frames can be transitory or permanent.

Transient interactions with the user can also take place in *menus* which can “pop-up” anywhere on the screen, and in *alerts*.

The run-time system is based on a central *Notifier* in each application which distributes input to the appropriate window, and a *window manager* which manages overlapping windows, distributing to the appropriate application.

The exchange of data between applications running in separate windows (in the same or separate processes) is facilitated by a *Selection Service*.

The Sun implementations of graphics standards — CGI, CORE, GKS — include extensions to run within windows. See the *SunCGI Reference Manual*, the *Sun-Core Reference Manual*, and the *SunGKS* manual, respectively, for more information.

History

Release 3.0

SunView first appeared in SunOS Release 3.0. It is an extension and refinement of SunWindows 2.0, containing many enhancements, bug fixes and new facilities not present in SunWindows. SunView is upward compatible with SunWindows — applications originally written under 2.0 can be recompiled and run under SunView.

In Release 3.0, these changes were reflected in a new organization for the SunView documentation. The material on Pixrects from the 2.0 *SunWindows Reference Manual* was broken out into a separate document, the *Pixrect Reference Manual*. Two new documents were introduced, the *SunView Programmer's Guide* and the *SunView System Programmer's Guide*.

The basic SunView interface, intended to meet the needs of simple and moderately complex applications, is documented here. This basic interface covers the functionality of the SunWindows window and tool layers.

The companion to this document is the *SunView System Programmer's Guide*. Its contents are a combination of new and old material. Several of its chapters document new facilities such as the Notifier, the Selection Service and the Defaults Package. Also included is material from the old *SunWindows Reference Manual* which is of interest to implementors of window managers and other advanced applications, such as the window manager routines.

Release 3.2

Many bug fixes and performance improvements were made to SunView for Release 3.2. This guide was extensively revised and added to for Release 3.2.

Release 3.4

Further bug fixes and enhancements came out with Release 3.4. These were documented in the *Release 3.4 Manual*.

Release 3.5

Release 3.5 brought support for hardware double-buffering under SunView and pixrects.

Release 4.0

Release 4.0 brings major enhancements to the SunView user interface — 'Search and Replace' in text subwindows, shadowed frames, 'Props' frame menu item, keyboard control of the caret, etc. — without involving major changes to its programmatic interface. For example, when programs that use text subwindows are recompiled, their users will be able to use the new 'Select Marked Text' pop-up frame. The *alerts* package is a new package for presenting information to the user and allowing him/her to make choices based on it.

This guide was revised and reprinted again for 4.0. The major changes are the addition of a new *Alerts* chapter and lists of attributes and functions at the beginning of some chapters as well as in the *SunView Interface Summary* chapter and *Index*.

On-Line Help

For information on the programmatic interface to the on-line help facilities of the Sun386i, see the *Sun386i Developer's Guide*. The spot help interface will be supported on all Sun workstations in the next release of SunView.

Code No Longer Supported

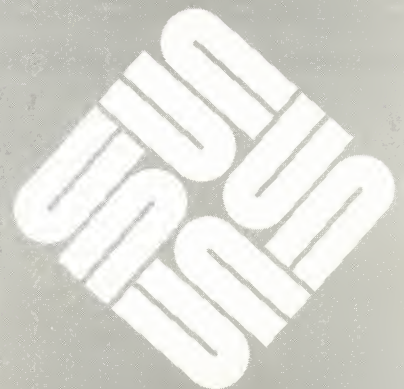
Do not use `DEFINE_ICON_FROM_IMAGE` or `DEFINE_CURSOR_FROM_IMAGE` as these macros may not be supported in future releases. Instead, use `icon_create()` and `cursor_create()` to create the icon or cursor at runtime. `icon_create()` is described in Chapter 14, *Icons*. `cursor_create()` is described in Chapter 13, *Cursors*.

The old SunWindows stacking menu package has been supplanted by the SunView walking menu package, described in Chapter 12 of this document. You should convert your applications to use the menu package, as the old package may not be included in future releases.

The new *alerts* package, described in Chapter 10, replaces use of the old (undocumented) `menu_prompt()` routine in situations where programs want to force the user to acknowledge a message or make a choice. Alerts are more flexible and easy-to-use than `menu_prompt()`, and we strongly encourage you to convert to them. Again, the old package may not be included in future releases.

The SunView Model

The SunView Model	9
2.1. Objects	9
Window Objects	11
Other Visual Objects	11
2.2. Examples of the use of Objects by Applications	12
2.3. Windows	16
Frames	16
Manipulating Frames Via Menus	18
Subwindows	19
2.4. Input: The Notifier	20
Callback Style of Programming	20
Why a Notification-Based System?	22
Relationship Between the Notifier, Objects, and the Application	22
Calling the Notifier Directly	24



The SunView Model

This chapter introduces the conceptual model presented by SunView, covering such basic concepts as *objects*, *windows* and the *Notifier*.

It is important that you understand the material in this chapter before you begin to write SunView applications.

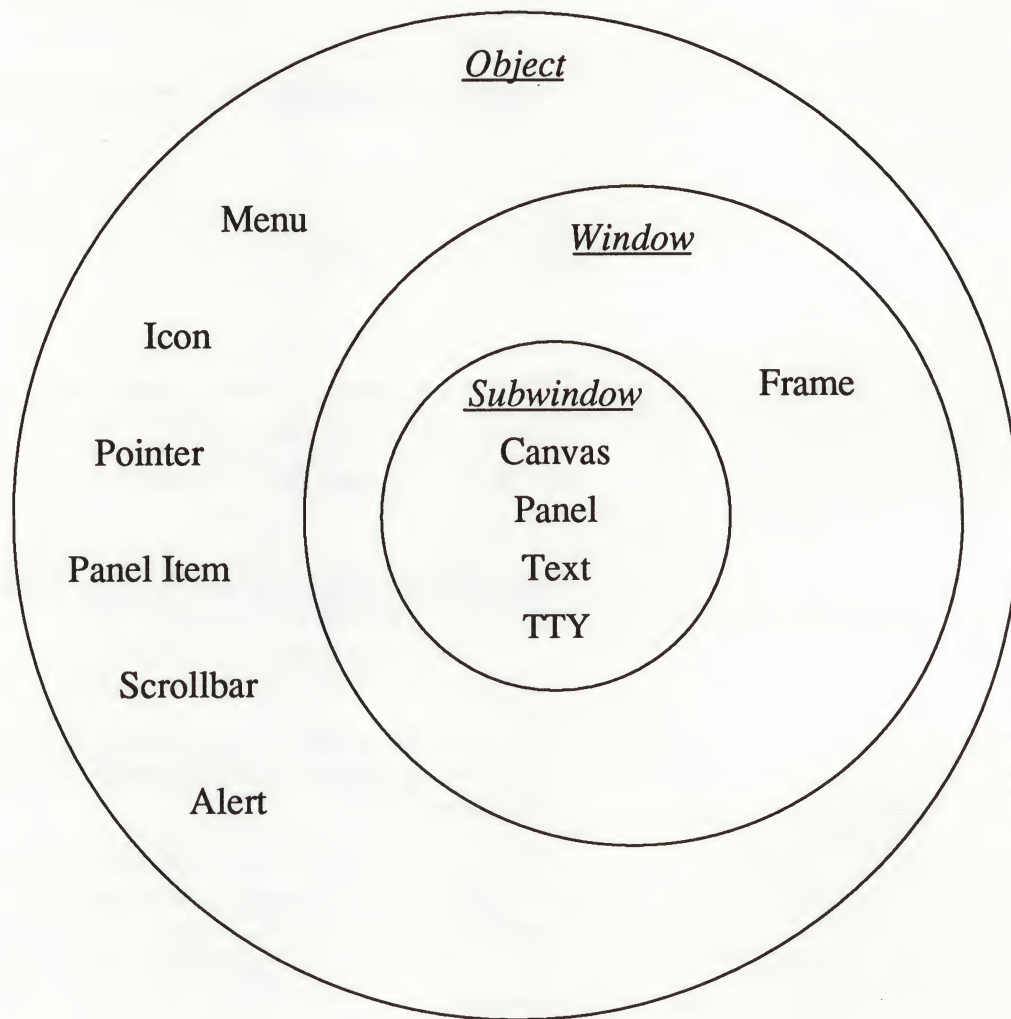
2.1. Objects

SunView is an *object-oriented* system. Think of SunView objects as visual building blocks which you use to assemble the user interface to your application. Different types of objects are provided, each with its particular properties; you employ whatever type of object you need for the task at hand.

The most important class of SunView objects are *windows*. Not all objects are windows, however. Other visual objects include cursors, icons, menus and scrollbars.

Technically, an object is a software entity presenting a functional interface. The implementation of the object is not exposed; you manipulate an object by passing its unique identifier, or *handle*, to its associated functions. The style of programmatic interface resulting from this object-oriented approach is outlined in this Chapter.

Figure 2-1 illustrates the different types and classes of SunView objects:

Figure 2-1 *SunView Objects*

The different types of objects are shown in normal font; the classes to which the objects belong are labeled in italics — *Subwindow*, *Window*, and *Object*.

Each object type is described briefly on the next page.

Window Objects

Window objects include *frames* and *subwindows*. Frames contain non-overlapping subwindows² within their borders. Currently, there are four types of subwindows provided by SunView:

- *Panel Subwindow* — A subwindow containing *panel items*.
- *Text Subwindow* — A subwindow containing text.
- *Canvas Subwindow* — A subwindow into which programs can draw.
- *TTY Subwindow* — a terminal emulator, in which commands can be given and programs executed.

The distinctions between frames and subwindows are explained in more detail in Section 2.3, *Windows*, later in this chapter.

Other Visual Objects

The other types of objects, like windows, are displayed on the screen, but they differ from windows in that they are less general and more tailored to their specific function. They include:

- *Panel Item* — A component of a panel that facilitates a particular type of interaction between the user and the application. Panel items can be moved, displayed or undisplayed under program control. There are several predefined types of items, including *buttons*, *message items*, *choice items*, *text items* and *sliders*.
- *Scrollbar* — An object attached to and displayed within a subwindow through which a user can control which portion of the subwindow's contents are displayed. Both vertical and horizontal scrollbars can be attached to panels and canvases. Text subwindows contain vertical scrollbars by default (they cannot contain horizontal scrollbars).
- *Menu* — An object through which a user makes choices and issues commands. By convention in SunView, menus pop up when the user presses the right mouse button. Like windows, menus appear on the screen when needed, and disappear when they have served their purpose. Menus, however, differ from windows in several ways. First, they are more ephemeral — a menu only remains on the screen as long as the menu button remains depressed,³ in contrast to a window, which remains on the screen until the user indicates he is done or the controlling program explicitly undisplays it. Second, menus are less flexible than windows; they are designed specifically to allow the user to choose from among a list of actions.

ephemeral = kortlevande

² It is SunView's window layout policy that enforces non-overlapping subwindows, not some limitation of the system. If you access the window system at a very low level, subwindows can overlap successfully.

³ The one exception is in the case of stay-up menus, which will appear when you click the RIGHT mouse button and disappear when you click it again.

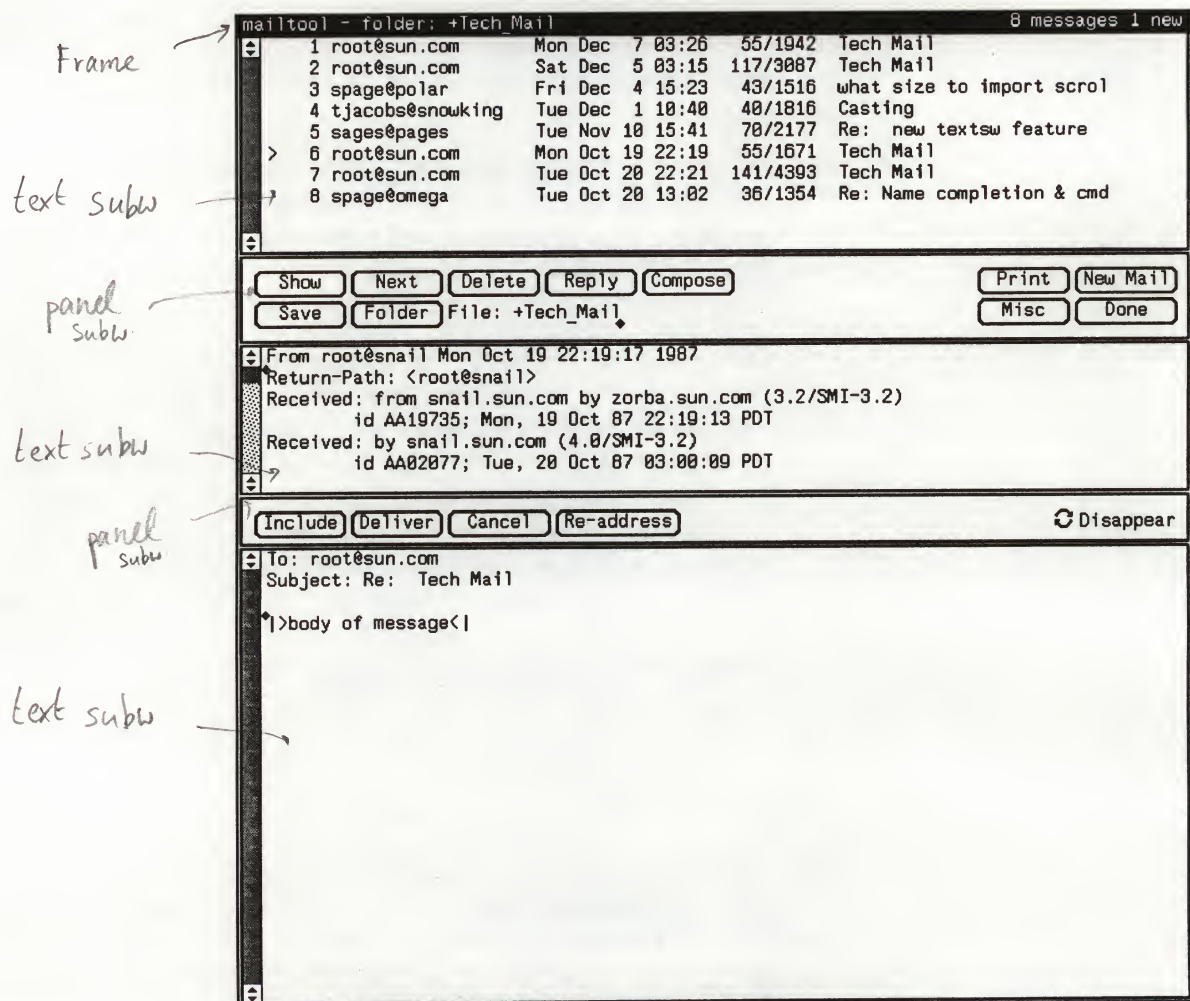
- *Alert* — a box on the screen which informs the user of some condition. It has one or more buttons which the user can push to dismiss the alert or choose a means of continuing. Like menus, alerts are ephemeral — they disappear as soon as the user pushes a button or otherwise dismisses the alert. Visually, they resemble simple panels containing only images, messages, and buttons.
- *Pointer* — The object indicating the mouse location on the screen.
- *Icon* — a small (usually 64 x 64 pixel) image representing the application.

The next section gives some examples showing how typical applications make use of SunView objects in their user interface.

2.2. Examples of the use of Objects by Applications

Figure 2-2 illustrates the `mailtool(1)`, which uses SunView objects to provide a mouse-oriented interface to the SunOS `mail(1)` program:

Figure 2-2 *Mailtool*

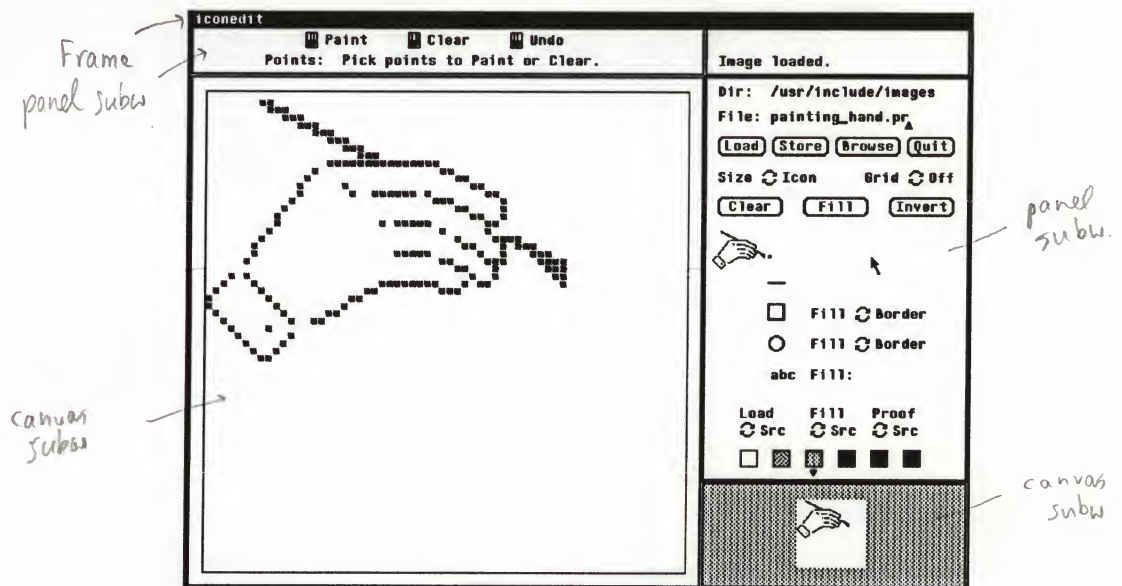


Mailtool consists of a frame containing three subwindows: a text subwindow in which the message headers are displayed, a panel containing various panel items (mostly buttons) through which the user can give commands to *mail*, and a text subwindow which displays the current message. An additional text subwindow and panel (shown in the figure) appear when you press the **reply** or **compose** buttons.

The text subwindows contain scrollbars, allowing the user to bring more information into view.

Figure 2-3 illustrates *iconedit*(1), a simple bitmap editor for generating images to be used by SunView applications:

Figure 2-3 *iconedit*



iconedit consists of a frame and five subwindows. From upper left to lower right they are:

- a panel containing instructions on how to use the mouse;
- a small panel for short messages;
- a canvas for drawing the image;
- a panel containing various items for issuing commands and setting options such as the size of the image being drawn, the drawing mode, etc;
- A small canvas for viewing the icon or cursor actual size.

None of these subwindows may be scrolled.

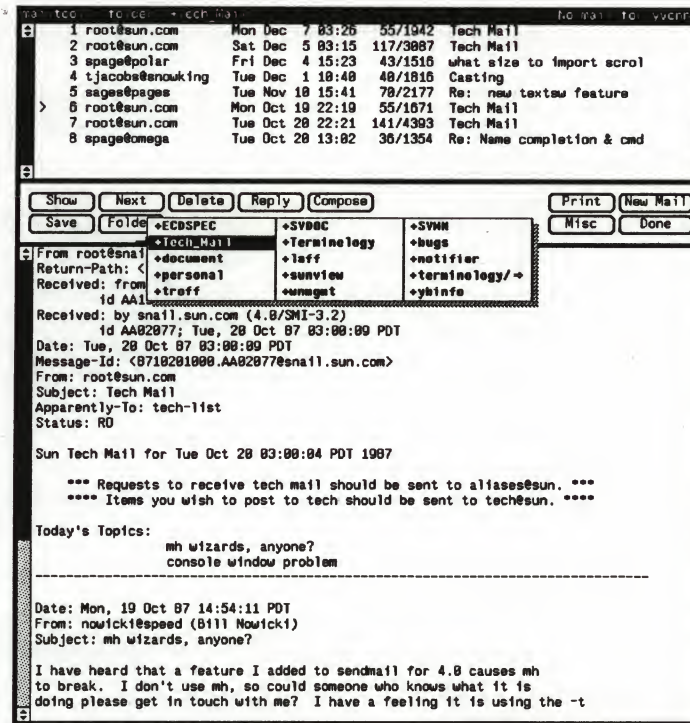
In Figure 2-4, the user has pushed the New Mail button, and the program brings up a hour glass cursor (in the upper right of the text subwindow) to denote that it is retrieving mail:

Figure 2-4 *iconedit-buttons*



In Figure 2-5, the user has pressed the mouse button over the Folder panel button in the panel:

Figure 2-5 *iconedit-menus*



mailtool has displayed a pop-up menu showing names of files which the user can insert into the text item File: by selecting a file. The purpose of this menu is to keep a current record of the mailfiles that the user has.

2.3. Windows

There are two basic classes of windows in SunView: overlapping *frames*, which contain non-overlapping *subwindows*. This section describes the distinction between the two.

Frames

A frame is not useful in itself — its purpose is to bring subwindows of different types together into a common framework so they can be operated on as a unit. A frame is said to *own* the subwindows it contains.

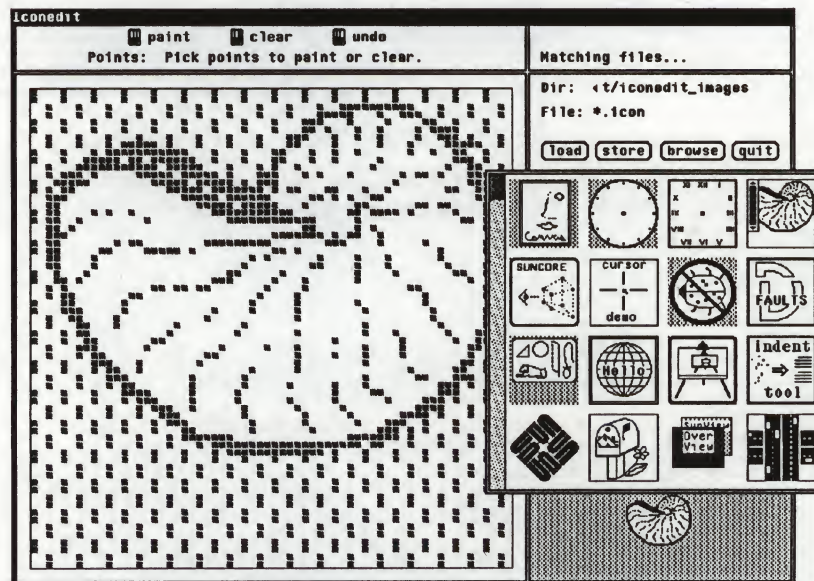
Frames may also own other frames. Thus the basic SunView structure is a hierarchy of windows. It could also be viewed as a tree of windows in which the non-leaf nodes are frames and the leaf nodes are subwindows.

The frame at the top of the hierarchy will be referred to in this document as the *base frame*; other frames will be referred to as *subframes*.⁴ Subframes are typically used to implement *pop-ups*, which perform auxiliary functions such as allowing the user to set options, or displaying help text.⁵

`iconedit` uses a pop-up for browsing images. When the user presses the button labeled **Browse**, `iconedit` displays a pop-up which consists of a subframe containing a single panel subwindow.

Figure 2-6 illustrates `iconedit` with its pop-up displayed.

Figure 2-6 A subframe



⁴ Note that while an application will usually be implemented as a single base frame (and its subwindows and subframes), it could well include several base frames.

⁵ For details on pop-ups, see Section 4.5.1, *Pop-ups*, in Chapter 4, *Using Windows*.

Figure 2-7 and Figure 2-8 illustrate the structures of *iconedit* and *mailtool* as a tree of windows. Frames are shown as rectangles; subwindows as circles:

Figure 2-7 *Structure of iconedit*

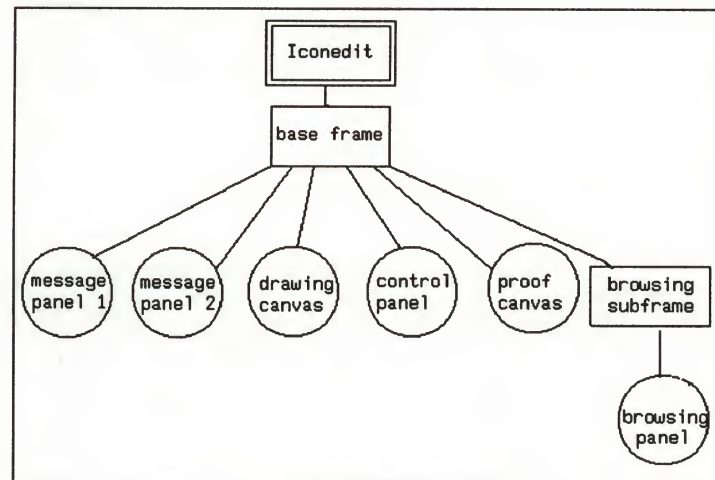
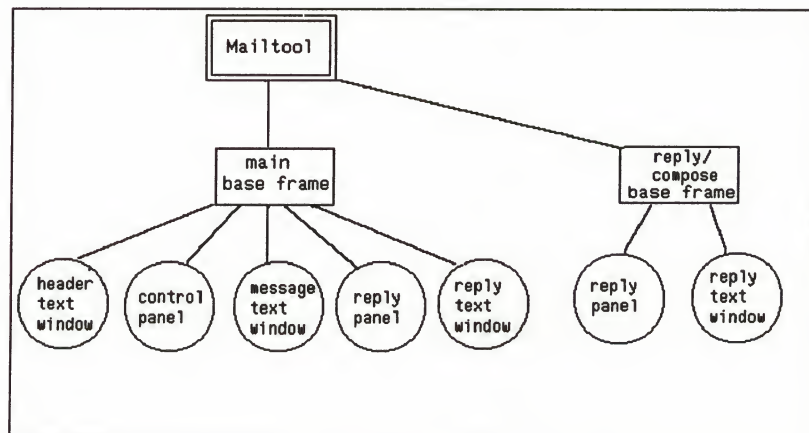


Figure 2-8 *Structure of mailtool*



Manipulating Frames Via Menus

Frames may be manipulated programmatically by setting the frame's *attributes*, as described in Chapter 4, *Using Windows*. Each frame also has a menu which allows the user to manipulate the frame directly. The frame menu is invoked by pressing the RIGHT mouse button on the exposed parts of the frame, which include the double lines surrounding the subwindows and the black *frame header* which usually appears at the top of the frame.

The menus for base frames and subframes differ slightly, as you can see from Figure 2-9 and Figure 2-10. The first window shows the base frame menu; the second window shows the subframe menu:

Figure 2-9 Base frame menu

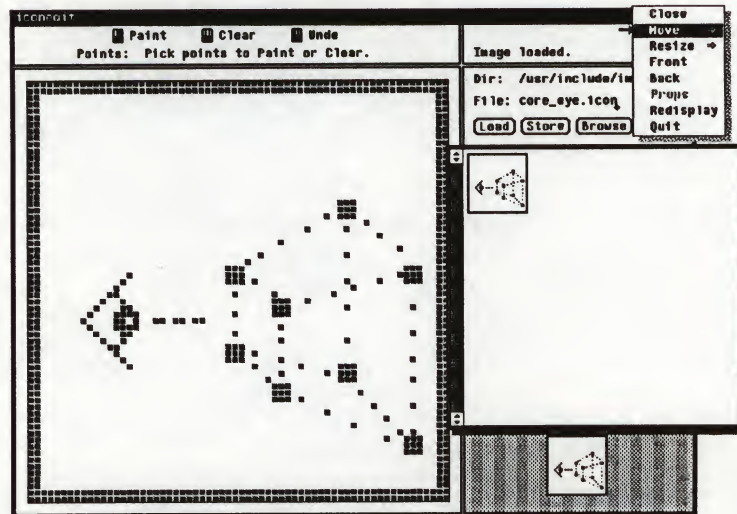
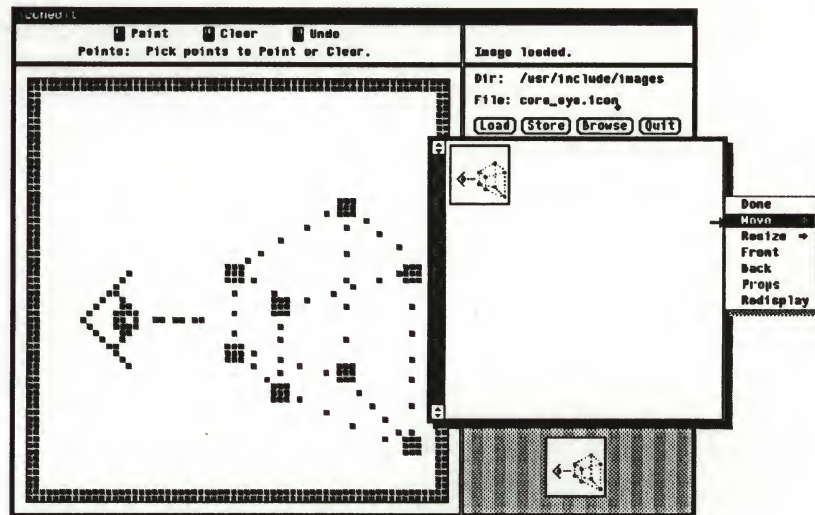


Figure 2-10 Subframe menu



Both menus contain the 'Move', 'Resize', 'Front', 'Back', and 'Redisplay' commands. 'Move' allows the user to change the frame's location. 'Resize' allows him or her to change the window's width and height. 'Front' causes the frame to move in front of the other windows, becoming fully visible on the "surface" of the screen, while 'Back' does the opposite, moving the frame behind any other windows occupying the same portion of the screen. 'Redisplay' simply causes the window to be displayed again.

When the user is finished working with a base frame he may want to destroy it for good, in which case he would choose 'Quit'. Or he may want to 'Close' the frame, with the anticipation of opening it later and continuing work where he left off. A base frame in its closed state is represented on the screen as a small (usually 64 by 64 pixel) *icon*. The icon is typically a picture indicating the function of the underlying application.

Subframes may not be closed into icons; when the user finishes with a subframe, he simply chooses *Done* from the menu. While not destroying the subframe, this causes it to disappear from the screen.

Subwindows

Subwindows differ from frames in several basic ways. Subwindows never exist independently. They are always owned by a frame, and may not themselves own subwindows or subframes. While frames can be moved freely around the screen, subwindows are constrained to fit within the borders of the frame to which they belong. Also in contrast to frames, subwindows are *tiled* — they may not overlap each other within their frame. Within these constraints (which are enforced by a run-time *boundary manager*) subwindows may be moved and resized by either a program or a user.

So far this chapter has discussed the static aspects of the SunView model. The section below outlines the system's model from a dynamic point of view.

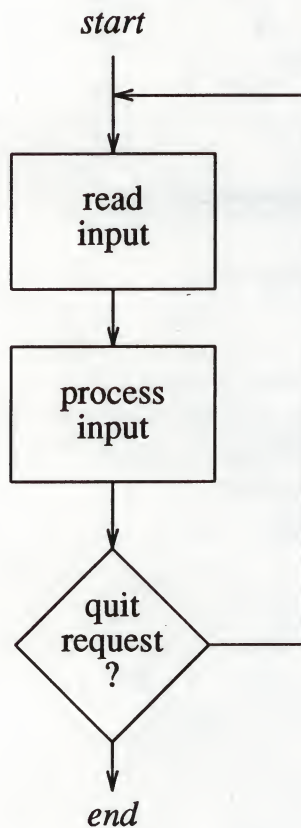
2.4. Input: The Notifier

SunView is a *notification-based* system. The Notifier acts as the controlling entity within a user process, reading UNIX input from the kernel, and formatting it into higher-level *events*, which it distributes to the different SunView objects.⁶

Callback Style of Programming

In the conventional style of interactive programming, the main control loop resides in the application. An editor, for example, will read a character, take some action based on the character, then read the next character, and so on. When a character is received that represents the user's request to quit, the program exits. Figure 2-11 illustrates this approach:

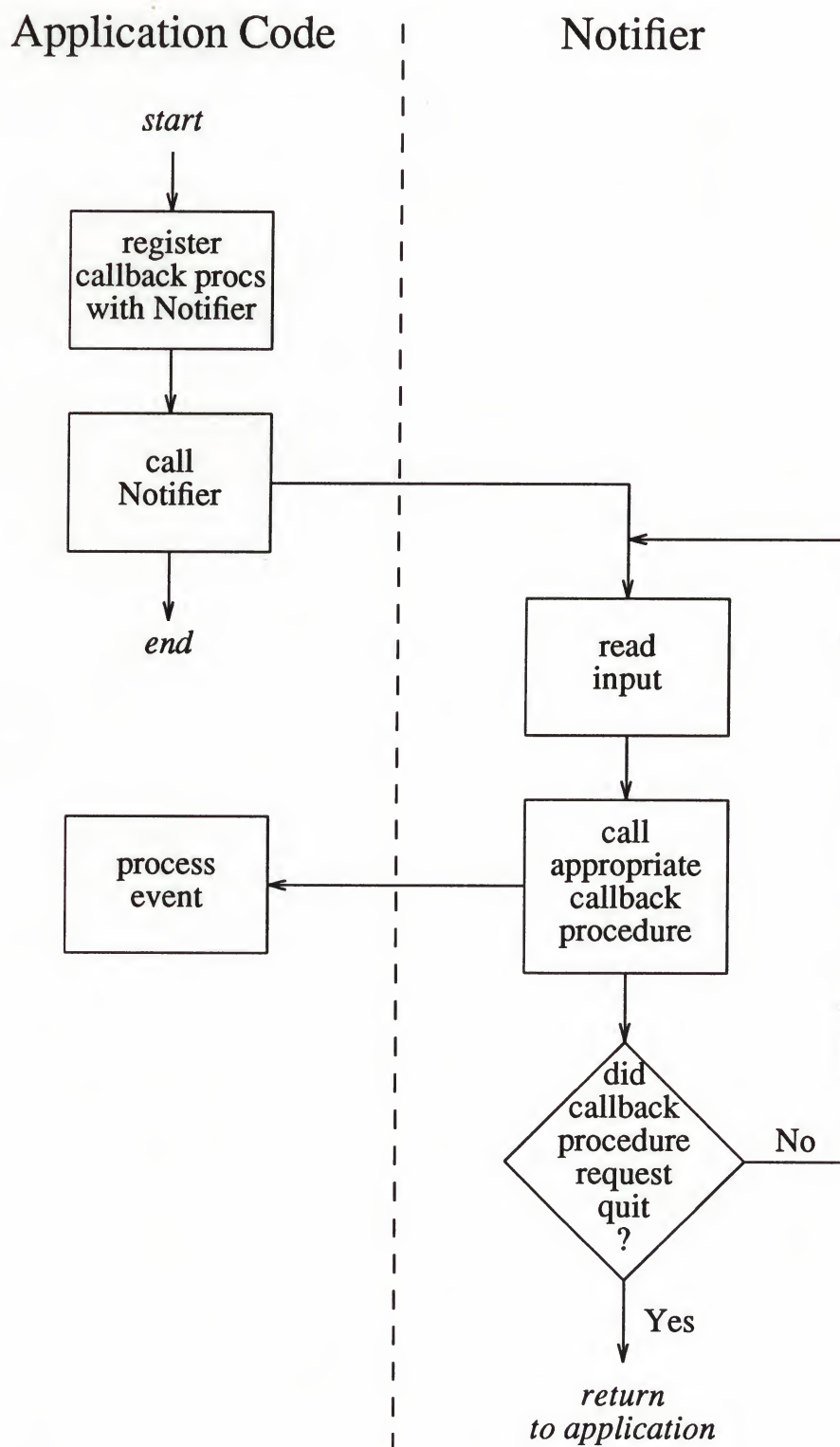
Figure 2-11 *Flow of Control in a Conventional Program*



Notification-based systems invert this “straight line” control structure. The main control loop resides in the Notifier, not the application. The Notifier reads events and *notifies*, or *calls out* to, various procedures which the application has previously registered with the Notifier. These procedures are called *notify procs* or *callback procs*. This control structure is shown in Figure 2-12.

⁶ SunView events are in a form which you can easily use: an ASCII key has been pressed, a mouse button has been pressed or released, the mouse has moved, the mouse has entered or exited a window, etc. Events are described in detail in Chapter 6, *Handling Input*.

Figure 2-12 Flow of Control in a Notifier-based Program



Why a Notification-Based System?

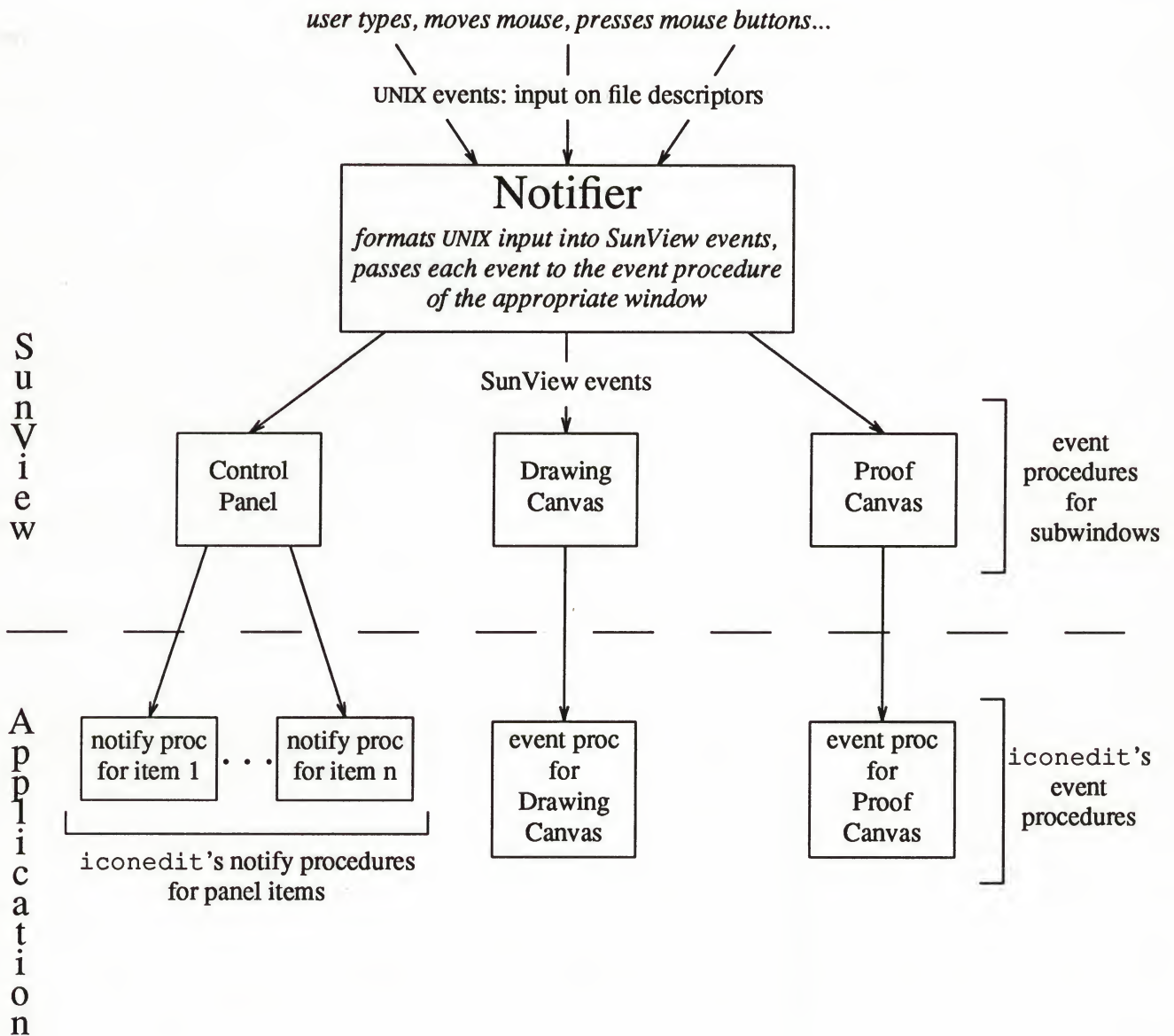
For programmers who are not used to it, this callback style of programming takes some getting used to. Its big advantage is that it takes over the burden of managing a complex, event-driven environment. In SunView, an application typically has many objects. In the absence of a centralized notifier, each application must be responsible for detecting and dispatching events to all the objects in the process. With a centralized Notifier, each component of an application receives only the events the user has directed towards it.

Relationship Between the Notifier, Objects, and the Application

It is not necessary for you to interact with the Notifier directly in your application. SunView has a two-tiered scheme in which the packages that support the various objects — panels, canvases, scrollbars, etc. — interact with the Notifier directly, registering their own callback procedures. The application, in turn, registers its own callback procedures with the object.

Typically, when writing a SunView application you first create the various windows and other objects you need for your interface, and register your callback procedures with the objects. Then you pass control to the Notifier. The work is done in the various callback procedures.

Let's illustrate the relationship of the Notifier, the SunView objects and the application by taking `iconedit` as an example. Figure 2-13 illustrates how the Notifier receives UNIX input and calls back to `iconedit`'s subwindows, which in turn call back to procedures supplied by `iconedit`.

Figure 2-13 Flow of Input Events in *iconedit*, a SunView Application

The main point of the diagram on the preceding page is to make clear the double-tiered callback scheme. How you register the callback procedures will be explained in the chapters on panels and canvases.

One point worth mentioning is the distinction between the “event procedures” for the canvases and the “notify procedures” for the panel items. They are all callback procedures, but they have different purposes. The canvas’s event procedure doesn’t do much work — basically it calls out to the application’s event procedure each time an event is received. The application sees every event and is free to interpret the events however it likes.

The event procedure for panels, on the other hand, does quite a bit of processing. It determines which item should receive the event, and places its own interpretation on events — the middle mouse button is ignored, left mouse button down over an item is interpreted as a “tentative” activation of the item, etc. It does not call back to the notify procedure for the item until it receives a left mouse button up over the item. So panel item notify procedures are not so much concerned with the event which caused them to be called, but with the fact that the button was pushed, or a new choice made, etc.

Calling the Notifier Directly

As mentioned previously, for many applications you will not need to call or be called by the Notifier directly — the Notifier calls back to the subwindows, which in turn call back to your application.

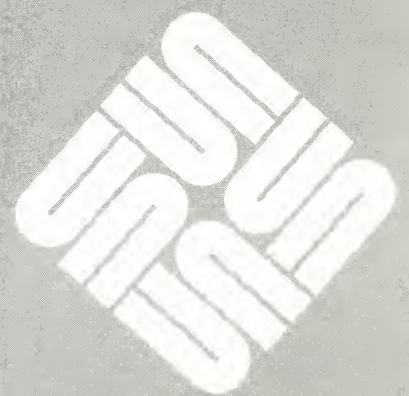
However, if you need to use signals, or be notified of the death of a child process which you have spawned, you do need to call the Notifier directly.

The Notifier also provides calls which allow you to insert your own routine in the event stream ahead of a window. This technique is known as *interposition*.

When and how to call the Notifier directly is covered in Chapter 17, *The Notifier*.

Interface Outline

Interface Outline	27
SunView Libraries	27
Compiling SunView Programs	27
Header Files	27
Object Handles	28
Attribute-based Functions	28
Standard Functions	29
Example of SunView-Style Programming	29
Attribute List Size	29
Reserved Namespaces	30



Interface Outline

This chapter outlines the SunView interface, the SunView libraries, header files, object handles, attributes and the standard functions applicable to objects of each type.

SunView Libraries

The SunView functions that an application calls are mostly in the library file `/usr/lib/libsuntool.a` if you are using the archive libraries and `/usr/lib/libsuntool.so` if you are using the shared libraries. These libraries include the code to create and manipulate high-level objects such as frames, panels, scrollbars and icons. These packages in turn call routines in `/usr/lib/libsunwindow.a` or `/usr/lib/libsunwindow.so` to create and manipulate windows and interact with the Notifier. These in turn call routines in `/usr/lib/libpixmap.a` or `/usr/lib/libpixmap.so` that do the drawing on the screen.

*NOTE Shared libraries are introduced in 4.0. The main benefit to using shared libraries is that the executables are much smaller (for example, 24K instead of 1Mb for `textedit` alone) because the libraries are loaded dynamically at runtime and are subsequently shared by other executables. Additionally, when the shared libraries are recompiled, new functionality is added, or bug fixes are made, the client applications don't need to be recompiled and linked unless the `.so` or an interface changed. For more information on shared libraries, see *Programming Utilities and Libraries*.*

Compiling SunView Programs

To compile a SunView program you must link in these three libraries, and, because they are built one on top of another, their order is important. For example, to compile a typical SunView application whose source is `myprog.c`, you would type in the command:

```
% cc -o myprog myprog.c -lsuntool -lsunwindow -lpixmap
```

Header Files

The basic definitions needed by a SunView application — covering windows, frames, menus, icons and cursors — are obtained by including the header file `<suntool/sunview.h>`. Definitions for the other types of object are found in their own include files — `<suntool/canvas.h>`, `<suntool/text.h>`, `<suntool/panel.h>`, etc.

Object Handles

When you create a SunView object, the creation function returns a *handle* for the object. Later, when you wish to manipulate the object or inquire about its state, you pass its handle to the appropriate function. This reliance on object handles is a way of *information-hiding*. The handles are *opaque* in the sense that you can't "see through" them to the actual data structure which represents the object.

Each object type has a corresponding type of handle. The window types of `Frame`, `Canvas`, `Textsw`, `Tty` and `Panel` are grouped under the type `Window`. So, for example, you can declare a panel as either a `Panel` or a `Window`, whichever is most appropriate. The other object types are `Panel_item`, `Menu`, `Scrollbar`, `Cursor`, and `Icon`.

Since C doesn't have an opaque type, all the opaque data types mentioned above are typedef'd to the UNIX type `caddr_t` (for "character address type"), which in turn is typedef'd to `char *`.

In addition to the opaque data types, there are several typedefs which refer not to pointers but to structs: `Event`, `Pixfont`, `Pixrect`, `Pixwin`, `Rect`, and `Rectlist`. Generally pointers to these structs are passed to SunView functions, so they are declared as `Event *`, `Pixwin *`, etc. The reason that the "*" is not included in the typedef is that the structs are publicly available, in contrast to the object handles, which include the "*" and which refer to structs that are not publicly available.

The SunView data types are summarized in the table beginning on page 324 in Chapter 19, *SunView Interface Summary*.

Attribute-based Functions

A model such as that used by SunView, which is based on complex and flexible objects, presents the problem of how the client is to manipulate the objects. The basic idea behind the SunView interface is to present a small number of functions, which take as arguments a large set of *attributes*.

For a given call to create or modify an object, only a subset of the set of all applicable attributes will be of interest. So that only the relevant attributes need be mentioned, SunView functions make use of variable-length *attribute lists*. An attribute list consists of attribute/value pairs, separated by commas, and ending with a zero.

Each type of object has its own set of attributes. The attributes have prefixes which indicate the type of object they apply to, i.e. `FRAME_*`, `TEXTSW_*`, `CANVAS_*`, `TTY_*`, `PANEL_*`, `MENU_*`, `CURSOR_*`, `ICON_*`, `SCROLL_*`, etc.

In addition to the sets of attributes applying to each type of object, there is a set of window attributes of the form `WIN_*` which apply to all window objects. These are attributes such as `WIN_HEIGHT` and `WIN_WIDTH`, which apply to all windows regardless of whether they happen to be panels, canvases, etc.

Standard Functions

For objects of all types there is a set of *standard functions* to create and destroy the object and to get and set the object's attributes.

Window functions are prefixed with `window_`, yielding

- `window_create()`,
- `window_get()`,
- `window_set()`, and
- `window_destroy()`.

Providing common window functions reduces the complexity of the interface. Non-window functions are prefixed with the name of the object. So, to take menus as an example, the standard functions are

- `menu_create()`,
- `menu_get()`,
- `menu_set()`, and
- `menu_destroy()`.

Example of SunView-Style Programming

The flavor of the interface is illustrated with the following code fragment, which creates a scrollbar with a width of 10 pixels and a black bubble. Later, the scrollbar's width is changed to 20 pixels. Finally, the scrollbar is destroyed:

```
Scrollbar bar;
bar = scrollbar_create(SCROLL_WIDTH, 10,
                      SCROLL_BAR_COLOR, SCROLL_BLACK,
                      0);
scrollbar_set(bar, SCROLL_WIDTH, 20, 0);
scrollbar_destroy(bar);
```

Note the zero which terminates the attribute lists in the `*_create()` and `*_set()` calls. The most common mistake in using attribute lists is to forget the final zero. This will not be flagged by the compiler as an error; however, it will cause SunView to generate a run-time error message.

Attribute List Size

As you can see from the example above, you can specify several attributes in a single `create()` or `set()` call. The maximum length of attribute lists in SunView is 250; see *Maximum Attribute List Size* in Chapter 18, *Attribute Utilities*.

Reserved Namespaces

SunView reserves names beginning with the object types, as well as certain other prefixes, for its own use.

The prefixes listed below should not be used by applications in lower, upper, or mixed case.

Table 3-1 *Reserved Prefixes*

ACTION_	icon_	scroll_
alert_	menu_	sels_
attr_	notify_	textsw_
canvas_	panel_	text_
cursor_	pixrect_	toolsw_
defaults_	pixwin_	tool_
ei_	pr_	ttysw_
es_	pw_	tty_
event_	rect_	window_
ev_	rl_	win_
frame_	scrollbar_	wmgr_
help_		

Using Windows

Using Windows	33
4.1. Basic Routines	35
Creating a Window	35
Initiating Event Processing	35
Modifying and Retrieving Window Attributes	35
Destroying Windows	36
4.2. Example 1— <i>hello_world</i>	37
4.3. Example 2— <i>simple_panel</i>	39
Some Frame Attributes	40
Panels	41
Fonts	41
Panel Items	41
Notify Procedure	41
Window Sizing — <code>window_fit()</code>	41
Fitting Frames Around Subwindows	41
4.4. Example 3— <i>lister</i>	42
4.5. Example 4— <i>filer</i>	44
Pop-ups	45
Pop-up Text Subwindow	45
Pop-up Property Sheet	46
Invoking the 'Props' Menu Item	46
<code>WIN_SHOW</code>	47
Pop-up Confirmer	47

window_loop	48
Restrictions on Pop-Up Frames	49
Controlling a Pop-up or Frame's Shadowing	49
4.6. Example 5— <i>image_browser_1</i>	50
Specifying Subwindow Size	50
Default Subwindow Layout	51
Explicit Subwindow Layout	51
Specifying Subwindow Sizes and Positions	52
Changing Subwindow Layout Dynamically	52
The Rect Structure	52
4.7. Example 6— <i>image_browser_2</i>	53
Row/Column Space	53
4.8. Attribute Ordering	55
Different Classes of Attributes	56
The Panel Package	56
4.9. File Descriptor Usage	57

Using Windows

This chapter describes how to build SunView applications out of frames and subwindows.

The first section presents the basic window routines. Succeeding sections give examples, ranging from the simplest possible application to a moderately useful file manager. For quick reference, the examples are given in the table below:

Table 4-1 *Window Usage Examples*

<i>Example</i>	<i>Description</i>	<i>Illustrates</i>	<i>Page</i>
<i>hello_world</i>	Minimal SunView program.	Compilation, frames.	37
<i>simple_panel</i>	Panel w/message and button.	Basic attributes, panels.	39
<i>lister</i>	Front end to <i>ls</i>	Panels, tty subwindows.	42
<i>filer</i>	File manager	Pop-ups, Selection Service.	44
<i>image_browser_1</i>	Displays images	Subwindow layout.	50
<i>image_browser_2</i>	Displays images	Row/column space.	53

Summary Listing and Tables

To give you a feeling of what you can do with frames and subwindows, the following page lists the available window and frame attributes, functions and macros. Many attributes are discussed as they occur in the examples, and in other chapters (use the *Index* to check). However, this chapter does not attempt complete coverage of all the attributes. All are briefly described with their arguments in the window and frame summary tables in Chapter 19, *SunView Interface Summary*:

- the *Window Attributes* table begins on page 379;
- the *Frame Attributes* table begins on page 382;
- the *Window Functions and Macros* table begins on page 384;
- the *Command Line Frame Arguments* table begins on page 386.

Window Attributes

WIN_BELOW	WIN_FIT_WIDTH	WIN_PERCENT_WIDTH
WIN_BOTTOM_MARGIN	WIN_FONT	WIN_PICK_INPUT_MASK
WIN_CLIENT_DATA	WIN_GRAB_ALL_INPUT	WIN_PIXWIN
WIN_COLUMNS	WIN_HEIGHT	WIN_RECT
WIN_COLUMN_GAP	WIN_HORIZONTAL_SCROLLBAR	WIN_RIGHT_MARGIN
WIN_COLUMN_WIDTH	WIN_IGNORE_KBD_EVENT	WIN_RIGHT_OF
WIN_CONSUME_KBD_EVENT	WIN_IGNORE_KBD_EVENTS	WIN_ROW_GAP
WIN_CONSUME_KBD_EVENTS	WIN_IGNORE_PICK_EVENT	WIN_ROW_HEIGHT
WIN_CONSUME_PICK_EVENT	WIN_IGNORE_PICK_EVENTS	WIN_ROWS
WIN_CONSUME_PICK_EVENTS	WIN_INPUT_DESIGNEE	WIN_SCREEN_RECT
WIN_CURSOR	WIN_KBD_FOCUS	WIN_SHOW
WIN_DEVICE_NAME	WIN_KBD_INPUT_MASK	WIN_TOP_MARGIN
WIN_DEVICE_NUMBER	WIN_LEFT_MARGIN	WIN_TYPE
WIN_ERROR_MSG	WIN_MENU	WIN_VERTICAL_SCROLLBAR
WIN_EVENT_PROC	WIN_MOUSE_XY	WIN_WIDTH
WIN_EVENT_STATE	WIN_NAME	WIN_X
WIN_FD	WIN_OWNER	WIN_Y
WIN_FIT_HEIGHT	WIN_PERCENT_HEIGHT	

Frame Attributes

FRAME_ARGS	FRAME_DEFAULT_DONE_PROC	FRAME_NO_CONFIRM
FRAME_ARGC_PTR_ARGV	FRAME_DONE_PROC	FRAME_NTH_SUBFRAME
FRAME_BACKGROUND_COLOR	FRAME_EMBOLDEN_LABEL	FRAME_NTH_SUBWINDOW
FRAME_CLOSED	FRAME_FOREGROUND_COLOR	FRAME_NTH_WINDOW
FRAME_CLOSED_RECT	FRAME_ICON	FRAME_OPEN_RECT
FRAME_CMDLINE_HELP_PROC	FRAME_INHERIT_COLORS	FRAME_SHOW_LABEL
FRAME_CURRENT_RECT	FRAME_LABEL	FRAME_SUBWINDOWS_ADJUSTABLE

Window Functions and Macros

window_bell(win)	window_get(win, attribute)
window_create(owner, type, attributes)	window_loop(subframe)
window_default_event_proc(window, event, arg)	window_main_loop(base_frame)
window_destroy(win)	window_read_event(window, event)
window_done(win)	window_refuse_kbd_focus(window)
window_fit(win)	window_release_event_lock(window)
window_fit_height(win)	window_return(value)
window_fit_width(win)	window_set(win, attributes)

4.1. Basic Routines

This section introduces the basic routines for using windows. It explains how to create, modify, and destroy windows.

Creating a Window

You create all windows with the function:

```
Window
window_create(owner, type, attributes)
    Window    owner;
    <window type> type;
    <attribute-list> attributes;
```

If you recall from Chapter 2, *The SunView Model*, a SunView application is implemented as a hierarchy of frames. Each frame owns one or more subwindows. The frame at the top of the hierarchy (the *base frame*) will have a null owner. In the above function, the *owner* parameter is the handle of the window to which the window returned by `window_create()` will belong. The *type* parameter is the type of the new window; for example, FRAME, PANEL, TEXTSW, CANVAS, or TTY.

A very simple example of this function would be to create a panel belonging to a frame called `base_frame`, you would use:

```
Panel panel;
window_create(base_frame, Panel, 0);
```

Initiating Event Processing

The `window_create()` call does not display the frame on the screen. You bring it to life after creating a base frame and its subwindows and subframes, by calling `window_main_loop(base_frame)`. This call displays the frame on the screen and begins processing the events by passing control to the Notifier. Chapter 2, *The SunView Model*, gave a brief explanation of the Notifier.

Keep in mind that subframes are treated different from base frames because they are not tied to the base frame that is activated in the `window_main_loop()` call. In addition, if you create a subframe with `WIN_SHOW` set to TRUE, when the user tries to manipulate the subframe 'garbage' data will appear on the screen.

Modifying and Retrieving Window Attributes

You modify and retrieve the value of window attributes with the following two functions:

```
int
window_set(window, attributes)
    Window window;
    <attribute-list> attributes;

caddr_t
window_get(window, attribute)
    Window window;
    Window_attribute attribute;
```

NOTE *If you call `window_get()` and specify an inappropriate attribute, a zero will be returned. For example, a sub frame cannot be closed. Therefore, the call `window_get(sub_frame, FRAME_CLOSED_RECT)` will not work, so the value returned will be zero. A segmentation violation will occur if an attempt is made to dereference the return value.*

When you get a pointer back from `window_get()`, the pointer points into a private data structure, whose contents may change.⁷

Destroying Windows

You destroy windows with the following two functions:

```
int
window_destroy(window)
    Window window;
```

```
int
window_done(window)
    Window window;
```

The difference between these two is that `window_destroy()` destroys only window and its subwindows and subframes. `window_done()`, on the other hand, destroys the entire hierarchy to which the subwindow or subframe belongs.

When `window_destroy()` is called on a window, the corresponding file descriptors cannot be used again until the Notifier is called. The file descriptor associated with the window is not reclaimed until the notifier has a chance to distribute notifications again.

The way `window_destroy()` works is that it asks the window owner if it is willing to be destroyed. If so, it queues up a notification procedure to destroy the window. This delay protects the program from destroying a window that is being accessed in the current call stack. You can work around this restriction, assuming you never reference this window again, by calling `notify_flush_pending()` after calling `window_destroy()`.

⁷ For most attributes the pointer returned by `window_get()` points into per-window storage, but for some the storage is static, per-process data. These attributes are flagged in the tables Chapter 19, *SunView Interface Summary*.

4.2. Example 1— *hello_world*

In learning a new programming language or environment, it usually helps to begin with a small program that simply prints some output. By creating, compiling, loading, and running the program, you will master the mechanical details. Here is a small SunView program:

```
#include <suntool/sunview.h>

main()
{
    Frame frame;
    frame = window_create(NULL, FRAME,
                          FRAME_LABEL, "hello world",
                          0);
    window_main_loop(frame);
}
```

After you create the above program in a file called `hello_world.c`, you compile it with the command:

```
% cc -o hello_world hello_world.c -lsuntool -lsunwindow -lpixrect
```

Where,

- `hello_world` is the executable output file that will be created
- `-lsuntool` specifies to link with the `suntool` object library
- `-lsunwindow` specifies to link with the `sunwindow` object library
- `-lpixrect` specifies to link with `pixrect` object library

After you compile the program, type “`hello_world`”, and the window will come up as shown in Figure 4-1 — a single frame with the words “hello world” in the frame header:

Figure 4-1 *Hello World Window*

This window is “alive” within the SunView user interface; it can be closed, moved, resized, hidden, etc. When closed, a default icon is displayed, which contains the text from the frame header.



4.3. Example 2— *simple_panel*

The next program is more complex than the first program. It creates a frame that contains a frame label and a panel that contains a panel button and a message. This program also includes an image that appears when the window closes down to an icon. Some basic attributes dealing with fonts, icons, help, error messages and parsing command-line flags are introduced.

```
#include <stdio.h>
#include <suntool/sunview.h>
#include <suntool/panel.h>
#include <suntool/icon.h>

static void quit_proc();
Frame frame;
Panel panel;
Pixfont *bold;
Icon icon;

static short    icon_image[] = {
#include <images/hello_world.icon>
};
mpr_static(hello_world, 64, 64, 1, icon_image);

main(argc, argv)
int argc; char **argv;
{
    bold = pf_open("/usr/lib/fonts/fixedwidthfonts/screen.b.12");
    if (bold == NULL) exit(1);

    icon = icon_create(ICON_IMAGE, &hello_world, 0);
    frame = window_create(NULL, FRAME,
        FRAME_LABEL,          "hello_world_panel",
        FRAME_ICON,           icon,
        FRAME_ARGS,           argc, argv,
        WIN_ERROR_MSG,        "Can't create window.",
        0);

    panel = window_create(frame, PANEL, WIN_FONT, bold, 0);

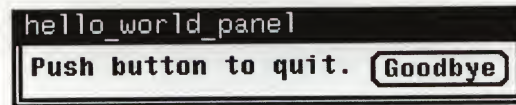
    panel_create_item(panel, PANEL_MESSAGE,
        PANEL_LABEL_STRING, "Push button to quit.", 0);
    panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel, "Good-bye", 0, 0),
        PANEL_NOTIFY_PROC, quit_proc,
        0);

    window_fit(panel);
    window_fit(frame);
    window_main_loop(frame);
}

static void
quit_proc()
{
    window_set(frame, FRAME_NO_CONFIRM, TRUE, 0);
    window_destroy(frame);
}
```

This program creates a frame containing a single panel with a message and a button:

Figure 4-2 *Hello World Panel*



The features and attributes used in the above program are discussed below.

Some Frame Attributes

The attributes are discussed below in the order that they appear in the above panel.

FRAME_LABEL

The string given as the value for `FRAME_LABEL` will appear in a black frame header strip at the top of the frame. If you do not want the label and the frame header to appear, then set the attribute `FRAME_SHOW_LABEL` to `FALSE`.

FRAME_ICON



The program used `FRAME_ICON` to specify the icon to be shown when the frame is closed. This is done by first using the macro `mpr_static()` to define a static memory pixrect that contains this data. Where `hello_world` is the name of the pixrect to be defined. The next three arguments specify the width, height, and depth of the image. Typically, for an icon, this is 64, 64, and 1. The final argument is an array of shorts that contains the bit pattern of the icon image. It takes its image from the file `/usr/include/images/hello_world.icon`. This statically defined image is passed to `icon_create()` at runtime.

The application uses `FRAME_ARGS`⁸ to pass command-line arguments given by the user to the frame. A set of command line arguments are recognized by all frames. These arguments allow the user to control such basic attributes as the frame's dimensions and label and whether the frame's initial state is open or closed, etc. These arguments begin with `-W`; for a complete list of them see the *Command Line Frame Arguments* table in Chapter 19, *SunView Interface Summary*.

WIN_ERROR_MSG

`WIN_ERROR_MSG` provides a simple form of error checking. If this attribute is not specified, then `window_create()` will return 0 on failure. If a value for `WIN_ERROR_MSG` is specified and `window_create()` fails, then it will print the error message on `stderr` and exit with a status of 1.

⁸ As an alternative to `FRAME_ARGS`, you can use `FRAME_ARGC_PTR_ARGV`, which takes a pointer to `argc`, rather than `argc` itself. This attribute causes `window_create()` to strip all arguments beginning with `-W` out of `argv`, and decrement `argc` accordingly.

Panels

The panel is created by calling `window_create()` with the previously created frame as the owner and `PANEL` as the window type.

Fonts

By default, text in the panel is rendered in the default system font, which `window_create()` obtains by calling `pf_default()`.⁹ The program specified a font by first opening the font with `pf_open()`, and then passing it into the panel as `WIN_FONT`.

NOTE *In the SunView context, note that setting `WIN_FONT` is not equivalent to specifying a font at run time with the `-Wt` command-line argument: `-Wt` opens the default system font, `WIN_FONT` doesn't. The only window types that currently make use of `WIN_FONT` to render characters are panels and text subwindows.*

Panel Items

The panel contains two panel items: the message saying "Push button to quit." and the **Good-bye** button. They are created with `panel_create_item()`.

Notify Procedure

The concept of *callback procedures* was introduced in Chapter 2, *The SunView Model*. Callback procedures for panel items are known as *notify procedures*.

The program registered its notify procedure `quit_proc()` with the **Quit** button using the attribute `PANEL_NOTIFY_PROC`. `quit_proc()` is called when the user selects the button. It in turn calls `window_destroy()`, which, as explained in the earlier subsection on *Destroying Windows*, causes `window_main_loop()` to return. Before calling `window_destroy()`, it disables the standard SunView confirmation by setting the attribute `FRAME_NO_CONFIRM` for the frame.

**Window Sizing —
`window_fit()`**

The final feature illustrated by the example is the use of the `window_fit()` macro. This macro causes a window to exactly fit its contents.

The contents of a panel are its panel items; the contents of a frame are its subwindows. Therefore, the example program calls `window_fit()` twice, first fitting the panel around its two items, then fitting the frame around its panel.

A `window_fit_width()` macro and a `window_fit_height()` macro are used to permit adjusting in only one dimension. These correspond to the window attributes `WIN_FIT_WIDTH` and `WIN_FIT_HEIGHT`.

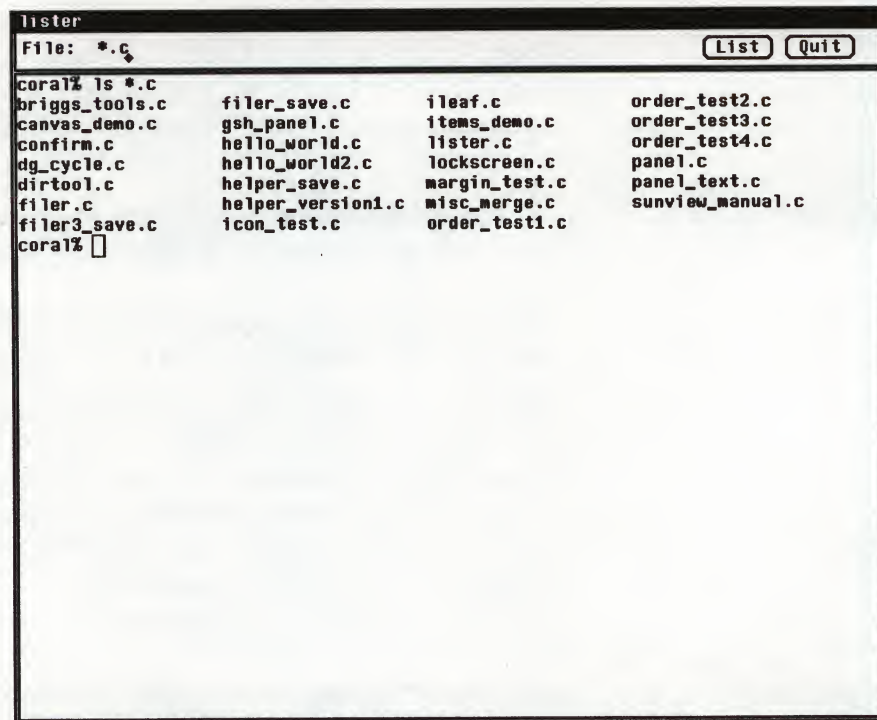
**Fitting Frames Around
Subwindows**

Since Release 3.2, if you use `window_fit()` or its variants for sizing the width and height of a frame, you need to be careful that the subwindows have some specified size, or they will be shrunk very small by the `window_fit()` call. Usually you give a subwindow a fixed size in one or both dimensions, or size it to be a percentage of the frame's size. The default size of a frame is that it encloses an area 34 rows by 80 columns in its default font.

⁹ For details on fonts see the *Pixrect Reference Manual*.

4.4. Example 3— *lister*

Figure 4-3 illustrates a program to help manage files. The first version simply lets the user list files in the current directory, forming a front-end to the `ls(1)` command:

Figure 4-3 *lister*

The tool presents two subwindows. The top subwindow is a control panel with a text item. It contains a place to specify the files to be listed, a **List** button, and a **Quit** button.

Below the control panel is a tty subwindow. When the user pushes the **List** button, the program constructs a command string consisting of the string `"ls "`, followed by the value of the **File:** item, followed by a newline, and inputs the command string to the tty subwindow by calling `ttysw_input()`.

The program is listed in its entirety below.

Notice that the frame, the panel and the tty subwindow are all declared as type `Window`. They could just as well have been declared as type `Frame`, `Panel` and `Tty`.


```

#include <suntool/sunview.h>
#include <suntool/panel.h>
#include <suntool/tty.h>

Window frame, panel, tty;
Panel_item fname_item;

static void ls_proc(), quit_proc();

main(argc, argv)
int argc; char **argv;
{
    frame = window_create(NULL, FRAME,
                          FRAME_ARGS, argc, argv,
                          FRAME_LABEL, "lister",
                          0);
    panel = window_create(frame, PANEL, 0);
    create_panel_items();
    tty = window_create(frame, TTY, 0);
    window_main_loop(frame);
    exit(0);
}

create_panel_items()
{
    fname_item = panel_create_item(panel, PANEL_TEXT,
                                   PANEL_LABEL_STRING, "File: ",
                                   PANEL_VALUE_DISPLAY_LENGTH, 55,
                                   0);

    panel_create_item(panel, PANEL_BUTTON,
                      PANEL_LABEL_IMAGE, panel_button_image(panel, "List", 5, 0),
                      PANEL_NOTIFY_PROC, ls_proc,
                      0);

    panel_create_item(panel, PANEL_BUTTON,
                      PANEL_LABEL_IMAGE, panel_button_image(panel, "Quit", 5, 0),
                      PANEL_NOTIFY_PROC, quit_proc,
                      0);

    window_fit_height(panel);
}

static void
ls_proc(/* ARGS UNUSED */)
{
    char cmdstring[256];

    sprintf(cmdstring, "ls %s\n", panel_get_value(fname_item));
    ttysw_input(tty, cmdstring, strlen(cmdstring));
}

static void
quit_proc(/* ARGS UNUSED */)
{
    window_destroy(frame);
}

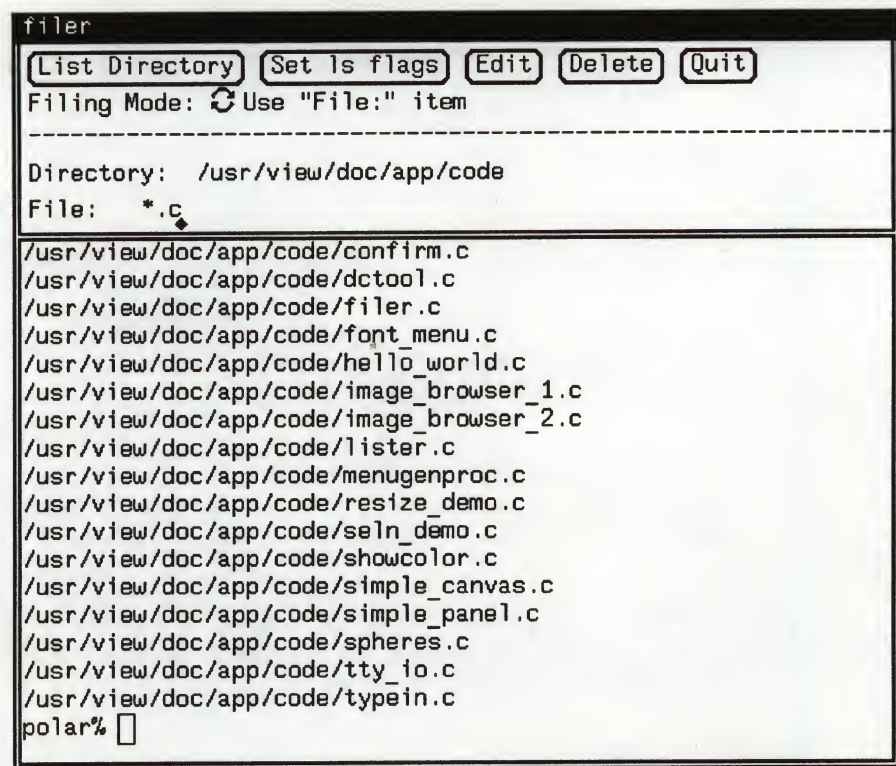
```


4.5. Example 4—*filer*

Our next example builds on the simple front end to `ls` given in the previous example to create a more interesting file manipulation tool. This application illustrates the use of the text subwindow, the Selection Service, and *pop-ups* — windows that appear on the screen and disappear dynamically during execution of a program.

In appearance, *filer* is similar to *lister*, in that it contains a control panel and tty subwindow. The user specifies the directory and file, and pushes the **List** button, causing the `ls` command to be sent to the tty subwindow:

Figure 4-4 *filer*



There are three new buttons, each of which illustrates a typical use of pop-ups:

Set ls flags a pop-up *property sheet* for setting options to `ls`;

Edit a pop-up text subwindow for browsing and editing files;

Delete a pop-up *confirmer* which forces the user to confirm or cancel.

The three buttons are discussed in the pages that follow. The discussion makes reference to specific routines in the *filer* program, which is listed in its entirety as *filer* in Appendix A, *Example Programs*.

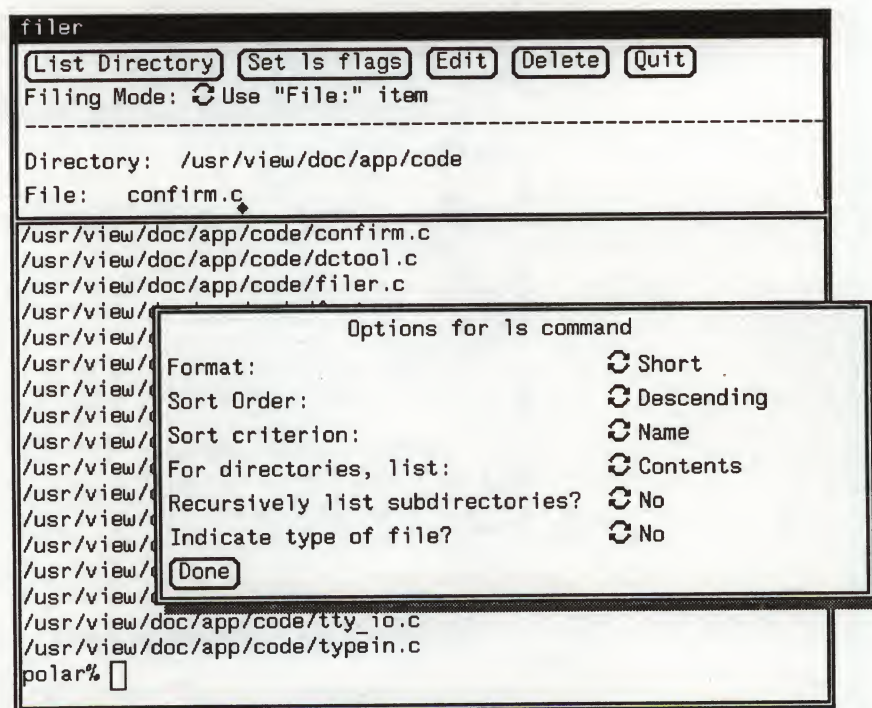
It then loads the file into the text subwindow, sets the frame header to the filename, and displays the frame with these two calls:

```
window_set(editsw, TEXTSW_FILE, filename, 0);
window_set(edit_frame, FRAME_LABEL, filename, WIN_SHOW, TRUE, 0);
```

Pop-up Property Sheet

The property sheet shown in Figure 4-6 is a typical example of a non-blocking pop-up. By pushing the **Set ls flags** button, the user can get a property sheet which allows him to set some of the options to the **ls** command. While the property sheet is displayed, the user can continue to interact with the application, setting options now and then. The user can cause the pop-up to disappear at any time by pushing the **Done** button, selecting *Done* from the subframe's menu, or by pressing the SunView function key labelled **Open**.

Figure 4-6 A Non-blocking Pop-up



Invoking the 'Props' Menu Item

Two attributes are used to control whether the 'Props' menu item is active or able to be invoked in the frame's menu. The code fragment given below is taken from the *filer* program.

The `FRAME_PROPS_ACTION_PROC` attribute specifies which procedure will be called when the 'Props' menu item is chosen or the **Props** key is pressed. In the code below, `FRAME_PROPS_PROC` specifies that the procedure `ls_flags_proc()` is called when the **Props** key is pressed.

The `FRAME_PROPS_ACTIVE` attribute specifies whether the procedure that is specified by the `FRAME_PROPS_ACTION_PROC` will be called or not. If the attribute `FRAME_PROPS_ACTIVE` is `TRUE`, then the frame menu will contain an un-greyed 'Props' menu item. If the attribute `FRAME_PROPS_ACTIVE` is `FALSE`, then the frame menu will contain a greyed out 'Props' menu item.

```
base_frame = window_create(NULL, FRAME,
                           FRAME_ARGS,      argc, argv,
                           FRAME_LABEL,      "filer",
                           FRAME_PROPS_ACTION_PROC, ls_flags_proc,
                           FRAME_PROPS_ACTIVE, TRUE,
                           0);
```

WIN_SHOW

The display of a non-blocking pop-up is controlled using the `WIN_SHOW` attribute. The initialization routine `create_ls_flags_popup()` creates the subframe, panel, and panel items for the property sheet. When the subframe is created, `WIN_SHOW` is `FALSE`.¹¹ The notify procedure for the Set ls flags button, `ls_flags_proc()`, simply sets `WIN_SHOW` to `TRUE` for the subframe.¹²

When the notify procedure for the List button, `ls_proc()`, is called, it calls `compose_ls_options()` to construct the appropriate string of flags based on the settings of the items in the property sheet.

Pop-up Confirmer

Both the property sheet and the editing subwindow described in the preceding section are examples of non-blocking pop-ups, in which the application continues to receive input while the pop-up is displayed.

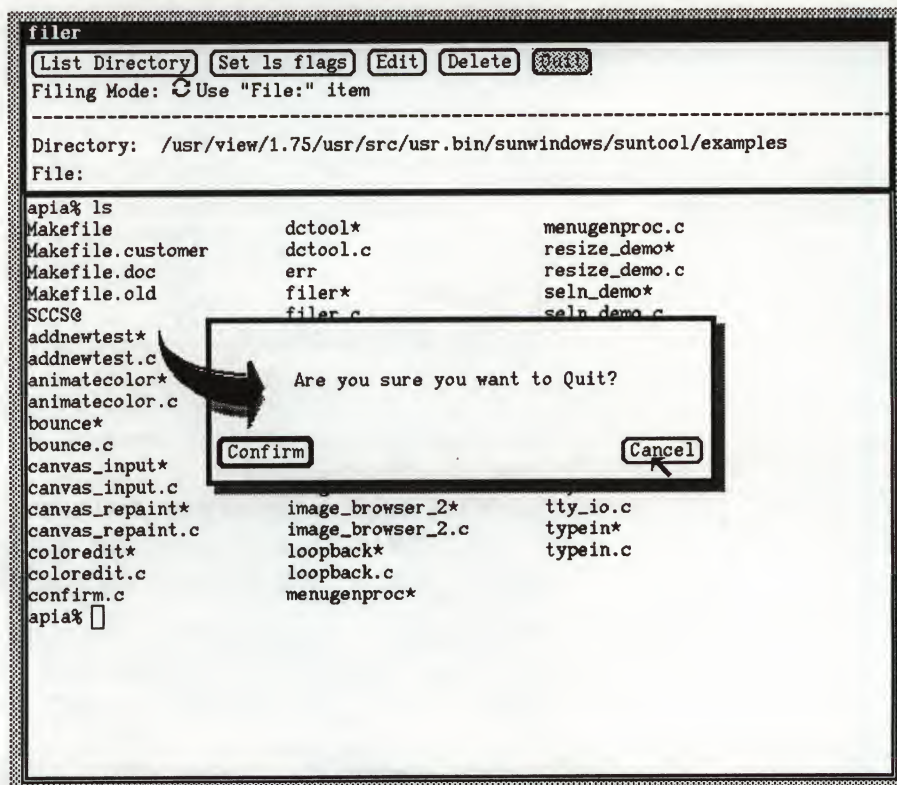
Blocking pop-ups differ in that, when displayed, they receive all input directed to the screen. Blocking pop-ups are appropriate when you want to force the user to confirm or cancel an irreversible operation before changing the application's state in any way.

Most uses of blocking pop-ups should use the alert package described in Chapter 10, *Alerts*. In the example given below, *filer* uses an alert for the Delete button confirmation. However, if you want to use other panel features, or other kinds of windows, then you can use `window_loop()` for the same effect.

For example, in Figure @NumberOf(alert-win), when the user makes a selection and pushes the Quit button, *filer* displays a pop-up asking for confirmation. All input is directed into this confirmer, and the user is forced to either accept the deletion by selecting Yes or cancel it by selecting No :

¹¹ Note that while `WIN_SHOW` defaults to `TRUE` for base frames, it defaults to `FALSE` for subframes. The same holds for `FRAME_SHOW_LABEL`.

¹² Note that the subframe won't actually be displayed until control is returned to the Notifier.

Figure 4-7 *Pop-up Confirmer*

window_loop

The display of a non-blocking pop-up is controlled using the WIN_SHOW attribute. The display of a blocking pop-up, on the other hand, is controlled with the two functions window_loop() and window_return().

```
caddr_t
window_loop(subframe)
    Frame subframe;

void
window_return(return_value)
    caddr_t return_value;
```

window_loop() causes the pop-up to be displayed and receive all input directed to the screen. The call will not return until window_return() is called from one of the pop-up's notify procedures. The value passed to window_return() as return_value will be returned by window_loop(). Its interpretation is up to the application. That is, it may be used to indicate whether the command was confirmed, whether a valid file name was entered, and so on.

Restrictions on Pop-Up Frames

There are some restrictions on pop-up frames displayed using `window_loop()`:

- You can only have one subwindow in the pop-up frame.
- The only subwindow types that work properly are canvases and panels.

These limitations do not apply to non-blocking pop-ups displayed using `WIN_SHOW`.

Controlling a Pop-up or Frame's Shadowing

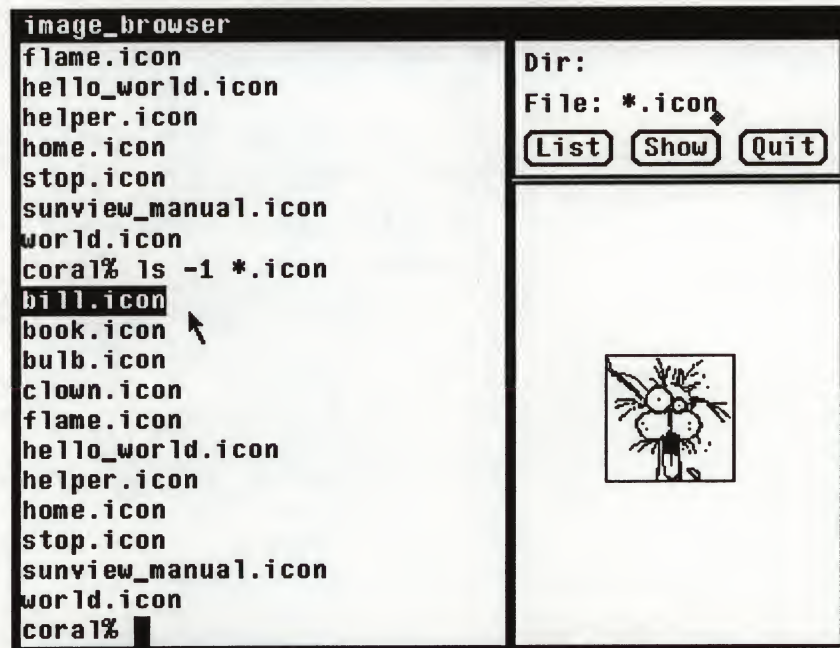
Sun's convention is that only transient items such as pop-ups have shadows. However, using the attribute `FRAME_SHOW_SHADOW` you may control the shadowing effect of a frame or a subframe:

- If you want your base frame to have a shadow, then set the attribute `FRAME_SHOW_SHADOW` to `TRUE`.
- You may stop a shadow from appearing with a `sub_frame` during create time by setting `FRAME_SHADOW` to `FALSE`.

4.6. Example 5— *image_browser_1*

Figure 4-8 illustrates how to specify the size and position of subwindows in order to get the layout that you want. This application lets the user view the images in files generated by *iconedit*. The user first presses the **List** button to get a listing. The user then selects a file that contains an image and press the **Show** button to view the image:

Figure 4-8 *image_browser_1*



This example presents a somewhat more complex subwindow layout: the *tty* subwindow has been moved to the left, the control panel to the upper right, and a panel for displaying the image added on the lower right.

Specifying Subwindow Size

You can specify the size of a subwindow either in pixels, with `WIN_HEIGHT` and `WIN_WIDTH` or in terms of rows and columns, with `WIN_ROWS` and `WIN_COLUMNS`.¹³ If its dimensions are not specified, then a subwindow will extend in the *y* direction to the bottom edge, and in the *x* direction to the right edge of the frame. In this case the subwindow's height and width will have the special value `WIN_EXTEND_TO_EDGE`,¹⁴ and will track the edge of the frame at run time, expanding or shrinking appropriately when the user resizes the frame.

Keep in mind that if you alter the size of a frame so that it exactly borders on a subwindow by calling `window_fit()`, the dimension of the subwindow that touches the frame will automatically become `WIN_EXTEND_TO_EDGE`.

¹³ Row/column space is discussed in the next example.

¹⁴ It is meaningless to set the width or height of a frame to `WIN_EXTEND_TO_EDGE`, and it will interfere with subwindow behavior.

Default Subwindow Layout

The default subwindow layout algorithm is simple. The first subwindow is placed at the upper left corner of the frame (leaving space for the frame's header and a border). If the width of the previously-created subwindow is fixed, not extend-to-edge, then the next subwindow is placed to the right of it. If the width of the previously-created subwindow *is* extend-to-edge, then the next subwindow is placed below it, at the left of the frame.

Explicit Subwindow Layout

This default layout algorithm handles only very simple topologies. SunView provides attributes that allow you to specify more complex layouts by explicitly positioning subwindows. You can position one subwindow relative to another by using `WIN_BELOW` and `WIN_RIGHT_OF`. These attributes take as their value the handle of the subwindow you want the new subwindow to be below or to the right of.

image_browser_1, pictured on the preceding page, illustrates the use of `window_fit()` along with explicit subwindow positioning to obtain a particular layout. The relevant calls are shown below:

```

tty = window_create(frame, TTY,
                    WIN_ROWS,    20,
                    WIN_COLUMNS, 30,
                    0);

control_panel = window_create(frame, PANEL, 0);

(create panel items...)

window_fit(control_panel);

display_panel = window_create(frame, PANEL,
                              WIN_BELOW,    control_panel,
                              WIN_RIGHT_OF, tty,
                              0);

window_fit(frame);

```

First the `tty` subwindow is created with a fixed height and width. Then the control panel is created, with no specification of origin or dimensions.

Since the width of the previous subwindow was fixed, the control panel is placed by default just to the right. After its items are created, the control panel is shrunk around its items in both dimensions with `window_fit()`.

Next, the display panel is created and explicitly positioned below the control panel and to the right of the `tty` subwindow. Both dimensions of the display panel default to `WIN_EXTEND_TO_EDGE`.

Finally, `window_fit()` is called to shrink the frame to the height of the `tty` window and the combined width of the `tty` window and the control panel.¹⁵

¹⁵ `window_fit()` causes the window to shrink until it encounters the first fixed border. Subwindows which are extend-to-edge don't stop the shrinking.

NOTE *One thing to watch out for is that WIN_BELOW only affects the subwindow's y dimension, and WIN_RIGHT_OF only affects the x dimension.*

Specifying Subwindow Sizes and Positions

You can also specify the origin of a subwindow in pixels using WIN_X and WIN_Y. The computations for these attributes take the borders and header of the frame into account, so that specifying WIN_X and WIN_Y of 0 will then result in the subwindow being placed correctly at the upper left corner of the frame.

The program *resize_demo*, listed in Appendix A, uses these attributes to lay out its subwindows in a non-standard manner.

Changing Subwindow Layout Dynamically

If you programmatically change the size or position of subwindows after you create them, then you must explicitly re-specify the origin of any subwindows that are below or to the right of the altered subwindows. This must be done even if you specified the positions of these other subwindows using relative position attributes, such as WIN_BELOW.

This step is necessary because subwindows are not automatically laid out again when the positions and sizes of other subwindows are changed. They are only laid out again if the frame changes size. When re-specifying the layout of the other subwindows, you *can* use relative position attributes such as WIN_BELOW.

The Rect Structure

The attributes WIN_X, WIN_Y, WIN_WIDTH and WIN_HEIGHT, taken together, define the rectangle occupied by a window. This rectangle is actually stored as a Rect struct, which you can get or set using the attribute WIN_RECT. The definition of a Rect, found in <sunwindow/rect.h>, is:¹⁶

```
typedef struct rect {
    short r_left;
    short r_top;
    short r_width;
    short r_height;
} Rect;
```

The Rect is the basic data structure used in SunView window geometry. Where complex shapes are required, they are built up out of groups of rectangles.¹⁷

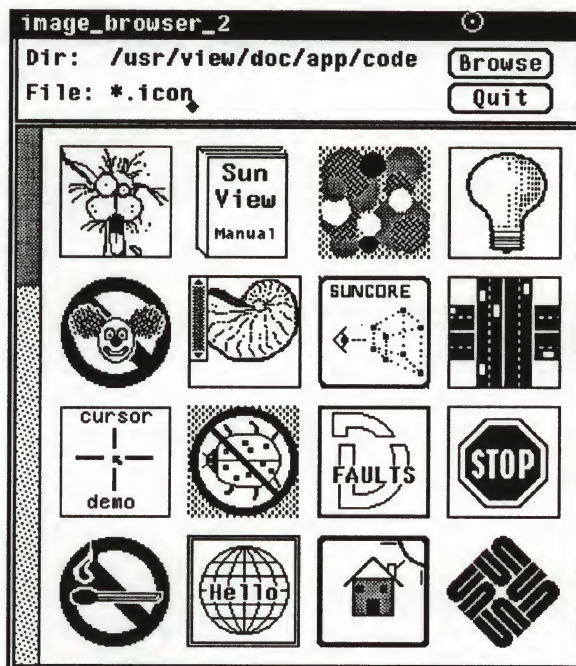
¹⁶ The result that a window returns is relative to a frame's *positioning space*. It is not self-relative and it is not parent-relative. Therefore, WIN_RECT should only be used for window positioning operations. Do not use it for pw_lock().

¹⁷ For a detailed discussion of rectangle geometry, including useful macros for operating on rectangles, see the chapter entitled *Rects and Rectlists* in the *SunView 1 System Programmer's Guide*.

4.7. Example 6— *image_browser_2*

In the next example, when the user specifies a filename and pushes **Browse** the images in the files are displayed in a scrollable panel:

Figure 4-9 *image_browser_2*



The point of this example is to illustrate how you can use *row/column space* to specify the size of a subwindow. The goal was to make the panel just the right size to display a single page of icons, with four rows, four columns, and 10 pixels of white space around each icon.

Row/Column Space

Row/column space refers to a logical grid defining the rows and columns of a window. You can define the row/column space for a window by using the attributes in the following table:

Table 4-2 *Window Row/Column Geometry Attributes*

<i>Attribute</i>	<i>Description</i>	<i>Default</i>	<i>Def. in Panels</i>
WIN_BOTTOM_MARGIN	Bottom margin.	0	(same)
WIN_COLUMN_GAP	Space after columns.	0	(same)
WIN_COLUMN_WIDTH	Width of a column.	Width of WIN_FONT.	(same)
WIN_LEFT_MARGIN	Left margin.	5	4
WIN_RIGHT_MARGIN	Right margin.	5	0
WIN_ROW_GAP	Space after rows.	0	5
WIN_ROW_HEIGHT	Height of a row.	Height of WIN_FONT	(same)
WIN_TOP_MARGIN	Top margin.	5 ¹⁹	4

¹⁹ In frames with headers, the default for WIN_TOP_MARGIN depends on the system font. With the default

Defining a Panel's Row/Column Space

Using the row/column space attributes, the icon browsing panel pictured on the preceding page is specified as follows:

```
Scrollbar scrollbar = scrollbar_create(SCROLL_MARGIN,10,0);
bar_width = (int)scrollbar_get(scrollbar, SCROLL_THICKNESS, 0);
display_panel = window_create(base_frame, PANEL,
                              WIN_VERTICAL_SCROLLBAR, scrollbar,
                              WIN_ROW_HEIGHT,          64,
                              WIN_COLUMN_WIDTH,         64,
                              WIN_ROW_GAP,              10,
                              WIN_COLUMN_GAP,           10,
                              WIN_LEFT_MARGIN,          bar_width + 10,
                              WIN_TOP_MARGIN,           10,
                              WIN_ROWS,                 4,
                              WIN_COLUMNS,              4,
                              0);
window_set(display_panel, WIN_LEFT_MARGIN, 10, 0);
```

This achieves our goal of a panel the right size for a 4x4 array of 64 pixel square icons, with 10 pixels of white space around each icon.

Positioning Panel Items in Row/Column Space

Once you have defined your row/column space, you can position panel items within that space with the `ATTR_ROW()` and `ATTR_COL()` macros.²⁰ The code fragment shown below shows how the items for the browsing panel are created and positioned at the proper row and column each time the **Browse** button is pushed:

```
for (row = 0, image_count = 0; image_count < files_count; row++)
  for (col = 0; col <= 4 && image_count < files_count; col++) {
    if (image = get_image(image_count)) {
      panel_create_item(display_panel, PANEL_MESSAGE,
                        PANEL_ITEM_Y,      ATTR_ROW(row),
                        PANEL_ITEM_X,      ATTR_COL(col),
                        PANEL_LABEL_IMAGE, image, 0);
      image_count++;
    }
  }
```

This example is complicated somewhat by an inconsistency in the way margins are handled in the current release of SunView. The left and top margins are used in two ways: for determining the size of the panel, and for determining the location of panel items positioned with `ATTR_COL()` and `ATTR_ROW()`. The size computation does not take into account any scrollbar which may be present; the positioning computation, on the other hand, does take the scrollbar into account. That is why, in the call to `window_create()` above, `WIN_LEFT_MARGIN` is set to the width of the scrollbar plus 10 pixels, and then set immediately afterward to 10 pixels.

system font, it defaults to 17.

²⁰ These "character unit macros" are described fully in Chapter 18, *Attribute Utilities*.

4.8. Attribute Ordering

The general rule is that attributes in SunView are evaluated in the order they are given. The following two examples of text subwindow calls illustrate how giving the same attributes in different orders can produce different effects:

```
window_set(textsw, TEXTSW_FILE, "file_1", 0);
window_set(textsw, TEXTSW_FIRST, 20, TEXTSW_FILE, "file_2", 0);

window_set(textsw, TEXTSW_FILE, "file_1", 0);
window_set(textsw, TEXTSW_FILE, "file_2", TEXTSW_FIRST, 20, 0);
```

In the first pair of calls, the index is first set to the 20th character of `file_1`, then `file_2` is loaded, starting at character zero. The second pair of calls first loads `file_2`, then sets the index in `file_2` to 20.

Command-line Arguments

The attribute `FRAME_ARGS` bears special mention. As described in the second example in this chapter, *simple_panel*, this attribute causes the frame to process the command-line arguments given by the user at run time. Some of these arguments correspond to attributes that can be set programmatically; for example, `-Wh` corresponds to `WIN_ROWS`.²¹

The basic rule, that attributes are evaluated in the order given, applies equally to attributes that are explicitly specified in the program and to those that are specified at run time using their command-line equivalents. If a given attribute is specified more than once, then the last setting is the one that takes effect. You can therefore control whether your application or the user has the last word by specifying attributes after or before `FRAME_ARGS`.

Let's take a couple of examples:

```
window_create(0, FRAME,
              FRAME_ARGS,  argc, argv,
              FRAME_LABEL, "LABEL FROM PROGRAM",
              WIN_ROWS,    10,
              0);

window_create(0, FRAME,
              FRAME_LABEL, "LABEL FROM PROGRAM",
              WIN_ROWS,    10,
              FRAME_ARGS,  argc, argv,
              0);
```

Assume that the program was invoked with a command line containing the following arguments:

```
-Wl "LABEL FROM COMMAND-LINE" -Wh 4
```

In the first call, by putting `FRAME_ARGS` at the start of the list, the application overrides the command-line arguments, and guarantees that the frame header will read "LABEL FROM PROGRAM" and the height will be 10 lines.

²¹ For a complete list of these arguments see the *Command Line Frame Arguments* table in Chapter 19, *SunView Interface Summary*.

In the second call, since `FRAME_ARGS` appears at the end of the list, the command-line arguments override what the application has specified, resulting in a label of "LABEL FROM COMMAND-LINE" and a height of 4 lines.

Keep in mind that if you specify `WIN_FONT`, it does not override the font that the user specified using `-Wt`.

Different Classes of Attributes

In the case of different objects, the window attributes (those beginning with `WIN_`) are processed after the others (`FRAME_*`, `PANEL_*`, and so on).

Suppose that you want to create a canvas with a scrollbar. You also want the logical canvas to expand when the user makes the window bigger, but never to shrink past its initial size, even if the user shrinks the window. The initial size of the canvas should be the size of the "inner" portion of the window — not including the scrollbar.

The straightforward approach would be to simply set all relevant attributes when the window is created, as in:

```
canvas = window_create(frame, CANVAS,
                       WIN_VERTICAL_SCROLLBAR, scrollbar_create(0),
                       CANVAS_AUTO_SHRINK, FALSE,
                       0);
```

This call, however, results in a canvas which is too big, extending underneath the vertical scrollbar. This is because of the order in which the `CANVAS_` and `WIN_` attributes are evaluated.

Since the window attributes are evaluated after the canvas attributes, the canvas size is set according to the initial size of the window, which does not have a scrollbar. By the time `WIN_VERTICAL_SCROLLBAR` is evaluated, the canvas refuses to shrink to the smaller inner portion of the window, since `CANVAS_AUTO_SHRINK` has already been evaluated and set to `FALSE`.

In general, you can force a particular order of evaluation by using separate `window_set()` calls, as in:

```
canvas = window_create(frame, CANVAS,
                       WIN_VERTICAL_SCROLLBAR, scrollbar_create(0),
                       0);
window_set(canvas, CANVAS_AUTO_SHRINK, FALSE, 0);
```

The Panel Package

The panel package deviates from the norm in that its attributes are generally not order-dependent. For example, you can specify the label of an item before the font, and the font will be used even though it appears after the label.

The only thing to watch out for is that you can't change the font in a single call, as in:

```

panel_set(text_item,
          PANEL_FONT,      font_1,
          PANEL_LABEL_STRING, "Label:",
          PANEL_FONT,      font_2,
          PANEL_VALUE,      "initial value",
          0);

```

The above call will cause both the label and the value for `text_item` to be rendered in `font_2`.

4.9. File Descriptor Usage

In SunView, each window is actually a device, `/dev/winnnn`, that is accessed through a file descriptor. Other packages such as the selection service also use file descriptors. In SunOS there is a limit to the number of file descriptors one program can have open; in Release 4.0 it is 64. Thus it is possible for your application to run out of file descriptors.

The following table summarizes how file descriptors are used in SunView.

Table 4-3 *SunView File Descriptor Usage*

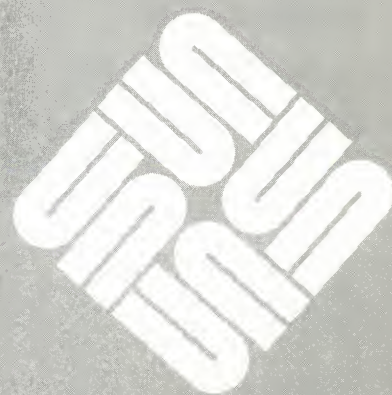
<i>Window Type/ Package</i>	<i>FD Usage</i>	<i>How FDs are used</i>
FRAME	1	1 for the window.
CANVAS	1	1 for the window.
TEXTSW	3 (2)	1 for the window, + 1 for the file to be edited (if any), + 1 for scratch (the <code>/tmp/Text...</code> file), 2 temporarily created during a save.
PANEL	1	1 for the window.
TTYSW	2	1 for the window, + 1 for the <i>pty</i> (pseudo-tty).
MENU	0	Fullscreen access uses the window's FD.
ALERT	1	1 for positioning Alerts have a frame and a panel; however, the FDs are allocated for the first alert and reused by subsequent alerts.
Pointer	0	Most pointers are managed by the kernel.
Icon	0	Frame uses same FD whether open or iconic.

Table 4-3 SunView File Descriptor Usage—Continued

<i>Window Type/ Package</i>	<i>FD Usage</i>	<i>How FDs are used</i>
Scrollbar	0	(implemented as a region -- read the <i>SunView System Programmer's Guide</i>)
<i>window manager</i>	(1)	1 temporarily used for window management operations.
UNIX	3	stdin/stdout/stderr
framebuffer	1	frame buffer FD gets allocated automatically with the base frame. The screen device must be opened for your program to draw on it.
Selection Service	3	selection service fd's are allocated whenever there is something that will set or get from the selection service. For example, if you put in selection service code or the first time a panel item is allocated.
	(1)	This uses sockets to communicate: 1 for the connection to the service + 1 to receive UDP requests + 1 TCP rendezvous socket for transfers. 1 transiently opened when a transfer is in progress to carry it.

Canvases

Canvases	61
5.1. Creating and Drawing into a Canvas	63
5.2. Scrolling Canvases	64
5.3. Canvas Model	65
The Canvas	65
5.4. Repainting	66
Retained Canvases	66
Non-Retained Canvases	66
The Repaint Procedure	66
Retained vs. Non-Retained	67
5.5. Tracking Changes in the Canvas Size	67
Initializing a Canvas	67
5.6. Automatic Sizing of the Canvas	69
5.7. Handling Input in Canvases	70
Default Input Mask	70
Writing Your Own Event Procedure	70
Translating Events from Canvas to Window Space	70
Border Highlighting	71
5.8. Color in Canvases	72
Setting the Colormap Segment	72
Color in Retained Canvases	72
Color in Scrollable Canvases	72



Canvases

The most basic type of subwindow provided by SunView is the *Canvas*. A canvas is essentially a window into which you can draw.

For a demonstration of the various canvas attributes, run the program `/usr/demo/canvas_demo`. For examples of canvases that illustrate event handling, run the image editor `iconedit(1)`. `iconedit` uses two canvases, the large drawing canvas on the left, and the small proof area on the lower right.

In order to use canvases you must include the header file `<suntool/canvas.h>`.

Summary Listing and Tables

To give you a feeling for what you can do with canvases, the following page lists the available canvas attributes, functions and macros. Many of these are discussed in the rest of this chapter and elsewhere (use the *Index* to check). All are briefly described with their arguments in the canvas summary tables in Chapter 19, *SunView Interface Summary*:

- the *Canvas Attributes* table begins on page 319;
- the *Canvas Functions and Macros* table begins on page 320.

Canvas Attributes

CANVAS_AUTO_CLEAR	CANVAS_FIXED_IMAGE	CANVAS_REPAINT_PROC
CANVAS_AUTO_EXPAND	CANVAS_HEIGHT	CANVAS_RESIZE_PROC
CANVAS_AUTO_SHRINK	CANVAS_MARGIN	CANVAS_RETAINED
CANVAS_FAST_MONO	CANVAS_PIXWIN	CANVAS_WIDTH

Canvas Functions and Macros

<code>canvas_event(canvas, event)</code>	<code>canvas_window_event(canvas, event)</code>
<code>canvas_pixwin(canvas)</code>	

5.1. Creating and Drawing into a Canvas

Like all windows in SunView, canvas subwindows are created with `window_create()`. When drawing into a canvas use the *canvas pixwin*, which you can get with the `canvas_pixwin()` macro.

The pixwin is the structure through which you render images in a window. You draw points, lines and text on a pixwin with a set of functions of the form `pw_*`() — `pw_write()`, `pw_vector()`, `pw_text()` etc.²²

Example 1:

As a beginning example, the following program puts up a canvas containing a box with the words "Hello World!":

```
#include <suntool/sunview.h>
#include <suntool/canvas.h>

main(argc, argv)
int  argc;
char **argv;
{
    Frame      frame;
    Canvas     canvas;
    Pixwin     *pw;

    /* create frame and canvas */
    frame = window_create(NULL, FRAME, 0);
    canvas = window_create(frame, CANVAS, 0);

    /* get the canvas pixwin to draw into */
    pw = canvas_pixwin(canvas);

    /* draw top, bottom, left, right borders of box */
    pw_vector(pw, 100, 100, 200, 100, PIX_SRC, 1);
    pw_vector(pw, 100, 200, 200, 200, PIX_SRC, 1);
    pw_vector(pw, 100, 100, 100, 200, PIX_SRC, 1);
    pw_vector(pw, 200, 100, 200, 200, PIX_SRC, 1);

    /* write text at (125,150) in default font */
    pw_text(pw, 125, 150, PIX_SRC, 0, "Hello World!");

    window_main_loop(frame);
    exit(0);
}
```

The `PIX_SRC` argument to `pw_vector()` and `pw_text()` is a *rasterop* function specifying the operation which is to produce the destination pixel values. There are several other rasterop functions besides `PIX_SRC`; they are described in Chapter 2 of the *Pixrect Reference Manual*.

²² Pixwins and their associated functions are covered in detail in Chapter 7, *Imaging Facilities: Pixwins*.

5.2. Scrolling Canvases

Many applications need to view and manipulate a large object through a smaller viewing window. To facilitate this SunView provides *scrollbars*, which can be attached to subwindows of type *canvas*, *text* or *panel*.

Example 2:

The code below creates a canvas that is scrollable in both directions:

```
frame = window_create(NULL, FRAME, 0);
canvas = window_create(frame, CANVAS,
    CANVAS_AUTO_SHRINK, FALSE,
    CANVAS_WIDTH, 1000,
    CANVAS_HEIGHT, 1000,
    WIN_VERTICAL_SCROLLBAR, scrollbar_create(0),
    WIN_HORIZONTAL_SCROLLBAR, scrollbar_create(0),
    0);
```

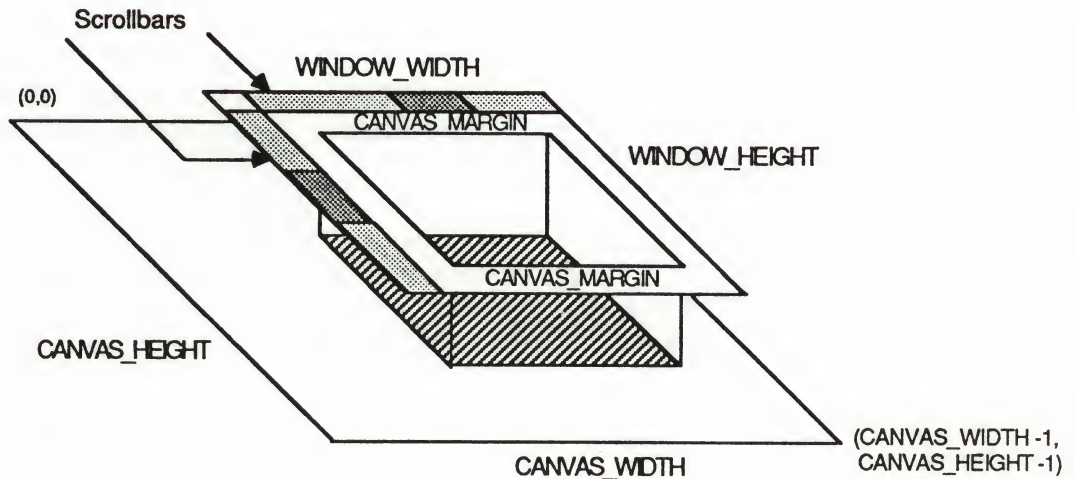
The distinction between the dimensions of the *canvas* and of the *window* is important. In the above example, we set the canvas width and height to 1000 pixels. Since the dimensions of the canvas subwindow (i.e. `WIN_WIDTH` and `WIN_HEIGHT`) were not explicitly set, the subwindow extends to fill the frame. The frame's dimensions, in turn, were not explicitly set, so it defaults to 25 lines by 80 characters in the default font. The result is a logical canvas roughly the area of the screen, which is viewed through a window about one fourth that size.

NOTE *It is necessary to explicitly disable the "auto-shrink" feature in the above example. If this were not done, the canvas size would be truncated to the size of the window. See Section 5.6, Automatic Sizing of the Canvas.*

5.3. Canvas Model

The components of a canvas subwindow and their relationships can be seen in Figure 5-1.

Figure 5-1 *Canvas Geometry*



The Canvas

Think of the canvas itself as a logical surface on which you can draw. The width and height of the canvas are set via the attributes `CANVAS_WIDTH` and `CANVAS_HEIGHT`. So the coordinate system is as shown in Figure 5-1, with the origin at the upper left corner and the point `(CANVAS_WIDTH-1, CANVAS_HEIGHT-1)` at the lower right corner. Note that the logical canvas origin is always at `(0, 0)`.

The Canvas Pixwin

As mentioned above, you draw on the canvas by writing into the canvas pixwin, which is retrieved via the `CANVAS_PIXWIN` attribute or the `canvas_pixwin()` macro.

The canvas pixwin is set up to take scrolling into account by performing the transformation from your canvas coordinate system to its pixwin coordinate system. So when you draw into the canvas pixwin using the `pw_*` functions you don't have to do any mapping yourself — the arguments you give should be in the canvas coordinate system.

Between the frame border and the canvas pixwin is a margin, set via the attribute `CANVAS_MARGIN`. This margin defaults to zero pixels, so in the simple case, the canvas pixwin occupies the entire inner area of the window pixwin. If one or more scrollbars are present, the canvas margin begins at the inside border of the scrollbar.

Note the distinction between the pixwin of the canvas (attribute `CANVAS_PIXWIN`) and the pixwin of the window (attribute `WIN_PIXWIN`). The canvas pixwin is one of several regions of the window's pixwin, which also includes the regions occupied by the scrollbars and the margin.

The canvas package manages the canvas pixwin for you. In particular, the clipping list is restricted to the area of the canvas pixwin actually backed by the canvas. This means that you can never draw off the edge of the canvas. For example, if you have set the canvas height to be less than the height of the canvas pixwin, any `pw_*` operations that attempt to draw below the canvas height will be clipped away.

5.4. Repainting

By default, canvases are *retained* — i.e. the canvas package maintains a copy of the bits on the screen in a *backing pixrect*, from which it automatically repaints the screen image when necessary. If you wish to handle repainting yourself, you can defeat this feature.

Retained Canvases

The canvas package allocates a backing pixrect the size of the logical canvas. When the canvas width or height changes, a new backing pixrect of the proper dimensions is allocated, the contents of the old pixrect are copied into the new pixrect, and the old pixrect is freed.

Non-Retained Canvases

For a non-retained canvas, set `CANVAS_RETAINED` to `FALSE`, and give your own repaint function as the value of `CANVAS_REPAINT_PROC`.

The repaint procedure is called whenever some part of the canvas has to be repainted onto the canvas pixwin. Note that if you supply a repaint proc, it will be called even if the canvas is retained — i.e. the canvas package will not automatically copy from the backing pixrect to the canvas pixwin.

The Repaint Procedure

The form of the repaint procedure is:

```
sample_repaint_proc(canvas, pixwin, repaint_area)
    Canvas      canvas;
    Pixwin      *pixwin;
    Rectlist    *repaint_area;
```

The first two arguments are the canvas and its pixwin (i.e. the value of `canvas_pixwin(canvas)`). The third argument, `repaint_area`, is a pointer to a list of rectangles (type `Rectlist *`) which define the area to be painted.²³

Before the canvas package calls your repaint procedure, it restricts the clipping list to the area which needs to be painted. Thus if your application is not capable of repainting arbitrary areas of the canvas you can repaint the entire image without worrying about excessive repainting.

If you choose not to redraw each individual rect in the repaint area, you can use the rectangle given by `repaint_area->rl_bound`, which is the bounding rectangle for the repaint area.

Note that if the attribute `CANVAS_AUTO_CLEAR` is `TRUE`, the canvas package will clear the repaint area before calling your repaint procedure.

²³ Rectlists are covered in detail in the chapter on *Rects and Rectlists* in the *SunView 1 System Programmer's Guide*.

Retained vs. Non-Retained

A retained canvas has two advantages. First, the repainting will be faster since it is a simple block copy operation. Second, it eliminates the need for the application to keep a display list from which to regenerate the image.

On the other hand there is a performance penalty on writing, since each operation is performed both on the canvas `pixwin` and the backing `pixrect`. This penalty may be reduced by using the `pw_batch()` call described in the chapter entitled *Imaging Facilities: Pixwins*.

5.5. Tracking Changes in the Canvas Size

The client's `resize` procedure is called whenever the canvas width or height changes. Its form is:

```
sample_resize_proc(canvas, width, height)
    Canvas    canvas;
    int       width;
    int       height;
```

NOTE *You should never repaint the image in the `resize` procedure, since if there is any new area to be painted, the `repaint` procedure will be called later.*

There are some subtle points to be aware of related to whether or not the image is fixed size (`CANVAS_FIXED_IMAGE` is `TRUE`). In the default case the image is fixed size, and the `repaint` procedure will not be called when the canvas gets smaller, since there will be no new canvas area to be repainted. If the image is *not* fixed size, then whenever the canvas size changes, the canvas package assumes that the entire canvas needs to be repainted, and the `repaint` area will contain the entire canvas.

Initializing a Canvas

Neither the `repaint` procedure nor the `resize` procedure will be called until the canvas subwindow has been displayed at least once. This allows you to create and initialize a canvas without having to deal with the `resize/repaint` procedures. The very first time the canvas is displayed, the `resize` procedure will be called with the current canvas size. This initial call to the `resize` procedure allows you to synchronize with the canvas size.

Example 3:

The canvas in the program below has a repaint procedure which fills the canvas with an appropriately sized rectangle and diagonals.

```
#include <suntool/sunview.h>
#include <suntool/canvas.h>

static void repaint_canvas();

main(argc, argv)
int    argc;
char **argv;
{
    Frame frame;

    frame = window_create(NULL, FRAME, 0);
    window_create(frame, CANVAS,
        CANVAS_RETAINED,    FALSE,
        CANVAS_FIXED_IMAGE, FALSE,
        CANVAS_REPAINT_PROC, repaint_canvas,
        0);
    window_main_loop(frame);
    exit(0);
}

static void
repaint_canvas(canvas, pw, repaint_area)
    Canvas    canvas;
    Pixwin    *pw;
    Rectlist  *repaint_area;
{
    int width  = (int>window_get(canvas, CANVAS_WIDTH);
    int height = (int>window_get(canvas, CANVAS_HEIGHT);
    int margin = 10;
    int xleft  = margin;
    int xright = width - margin;
    int ytop   = margin;
    int ybottom = height - margin;

    /* draw box */
    pw_vector(pw, xleft, ytop, xright, ytop, PIX_SRC, 1);
    pw_vector(pw, xright, ytop, xright, ybottom, PIX_SRC, 1);
    pw_vector(pw, xright, ybottom, xleft, ybottom, PIX_SRC, 1);
    pw_vector(pw, xleft, ybottom, xleft, ytop, PIX_SRC, 1);

    /* draw diagonals */
    pw_vector(pw, xleft, ytop, xright, ybottom, PIX_SRC, 1);
    pw_vector(pw, xright, ytop, xleft, ybottom, PIX_SRC, 1);
}
```

There are several points to note from the example on the previous page. First, since the width and height of the canvas are not specified, they default to the width and height of the window. Second, since the image being drawn is dependent on the size of the canvas, we set `CANVAS_FIXED_IMAGE` to `FALSE`. Third, when the repaint proc is called, we don't bother to draw the specified repaint area, instead we rely on the clipping list to be restricted correctly and simply redraw the entire image.

5.6. Automatic Sizing of the Canvas

Two attributes requiring some explanation are `CANVAS_AUTO_EXPAND` and `CANVAS_AUTO_SHRINK`. Setting both these attributes to `TRUE` allows you to have a drawing area which automatically tracks the size of the window.

If `CANVAS_AUTO_EXPAND` is `TRUE`, the canvas width and height are never allowed to be less than the edges of the canvas `pixwin`. For example, if you try to set `CANVAS_WIDTH` to a value which is smaller than the width of the canvas `pixwin`, the value will be automatically expanded (rounded up) to the width of the canvas `pixwin`.

The main use of `CANVAS_AUTO_EXPAND` is to allow the canvas to grow bigger as the user stretches the window. For example, if the canvas starts out exactly the same size as the canvas `pixwin`, and the user stretches the window, the canvas `pixwin` will get bigger, which will cause the canvas itself to expand.

Another point to keep in mind is that whenever you set `CANVAS_AUTO_EXPAND` to `TRUE`, the canvas will be expanded to the edges of the canvas `pixwin` (if it is smaller to begin with).

`CANVAS_AUTO_SHRINK` is symmetrical to `CANVAS_AUTO_EXPAND`. If `CANVAS_AUTO_SHRINK` is `TRUE`, the canvas width and height are never allowed to be greater than the edges of the canvas `pixwin`.

NOTE *As described in Section 4.8, Attribute Ordering, the canvas attributes are evaluated before the generic window attributes. This means that, if you want to set the window size and then disable automatic sizing of the canvas, you must first set the window size, then, in a separate `window_set()` call, disable `CANVAS_AUTO_SHRINK` and/or `CANVAS_AUTO_EXPAND`. If you do both in the same call, the auto-sizing will be turned off before the window size is set, so the canvas size will not match the window size you specify. Here is an example of how to do it correctly:*

```
canvas = window_create(frame, CANVAS,
                        WIN_HEIGHT, 400,
                        WIN_WIDTH, 600,
                        0);

window_set(canvas,
           CANVAS_AUTO_SHRINK, FALSE,
           CANVAS_AUTO_EXPAND, FALSE,
           0);
```


5.7. Handling Input in Canvases

Default Input Mask

This section gives some hints on basic handling of input in canvases.²⁴

By default, canvases enable LOC_WINENTER, LOC_WINEXIT, LOC_MOVE and the three mouse buttons, MS_LEFT, MS_MIDDLE and MS_RIGHT.²⁵

NOTE *Since the canvas pixwin is actually a region of the subwindow's pixwin, your event procedure will receive LOC_RGNENTER and LOC_RGNEXIT events rather than LOC_WINENTER and LOC_WINEXIT. The locator motion events — LOC_MOVE, LOC_STILL, LOC_DRAG, and LOC_TRAJECTORY — will only be passed to your event procedure if they fall within the canvas pixwin.*

You can enable events other than those listed above with the window attributes applying to events. So, for example, you could allow the user to type in text to a canvas by calling:

```
window_set(canvas, WIN_CONSUME_KBD_EVENT, WIN_ASCII_EVENTS, 0);
```

An application needing to track mouse motion with the button down would enable LOC_DRAG by calling:

```
window_set(canvas, WIN_CONSUME_PICK_EVENT, LOC_DRAG, 0);
```

Writing Your Own Event Procedure

If you supply an event procedure as the value of WIN_EVENT_PROC, it will get called when any event is received for the canvas. Before your event procedure gets called, however, the canvas package does some processing. If the event is WIN_REPAINT or WIN_RESIZE, the canvas package calls your repaint or resize procedures if necessary. If the event is SCROLL_REQUEST, then the canvas package performs the scroll.²⁶ The repaint, resize and scroll events are then passed to your event procedure. In the case of events which have x-y coordinates, the canvas package translates the events from the coordinate space of the canvas pixwin to that of the logical canvas.

Translating Events from Canvas to Window Space

Functions are provided to translate event coordinates from the coordinate space of the canvas to the coordinate space of the canvas subwindow, and *vice versa*.

To go from canvas space to window space, use canvas_window_event(). Keep in mind that the canvas_window_event function changes fields in its event argument structure. For example, if you want to put up a menu in a canvas

²⁴ The general input paradigm for Sunview is discussed in Chapter 6, *Handling Input*. See that chapter for a full discussion of the available input events and how to use them.

²⁵ Note that the canvas package expects to receive these events, and will not function properly if you disable them. Also, if the user has the enabled the *Left_Handed* option in the *Input* category of defaultsedit(1), the mouse buttons are reversed: MS_LEFT refers to the right mouse button, MS_RIGHT to the left mouse button.

²⁶ If you want write a procedure which is called *before* the repaint, resize or scroll event is processed by the canvas package, in order to modify the interpretation of the event, you must *interpose* on the event, as described in Chapter 17, *The Notifier*.

subwindow, you need to specify the menu's location in the coordinate of the subwindow, not of the canvas.

To go from window space to canvas space, use `canvas_event()`. This returns the `Event` * it is passed, with the `x` and `y` fields changed. The translation is necessary if you read your own events with `window_read_event()`, described in the next chapter, *Handling Input*.

Border Highlighting

The SunView convention is that a subwindow indicates that it is accepting keyboard events by highlighting its border. By default, canvas subwindows do not enable any keyboard events, so the border is not highlighted. However, if you explicitly enable keyboard events, by consuming `WIN_ASCII_EVENTS`, the canvas package will highlight the canvas border when it is given the input focus.

Example 4:

The program below prints out the corresponding string when the user types `0`, `1`, or `2` into its canvas:

```
#include <suntool/sunview.h>
#include <suntool/canvas.h>

static void my_event_proc();

main(argc, argv)
int    argc;
char **argv;
{
    Frame frame;

    frame = window_create(NULL, FRAME, 0);
    window_create(frame, CANVAS,
        WIN_CONSUME_KBD_EVENT, WIN_ASCII_EVENTS,
        WIN_EVENT_PROC,      my_event_proc,
        0);
    window_main_loop(frame);
    exit(0);
}

static void
my_event_proc(canvas, event)
    Canvas canvas;
    Event *event;
{
    char *string = NULL;

    switch (event_action(event)) {
        case '0':
            string = "zero";
            break;

        case '1':
            string = "one ";
            break;
    }
}
```

```

    case '2':
        string = "two ";
        break;

    default:
        break;
}
if (string != NULL)
    pw_text(canvas_pixwin(canvas),
            10, 10, PIX_SRC, NULL, string);
}

```

5.8. Color in Canvases

You can use color in canvases by specifying a colormap segment for the canvas with the colormap manipulation routines described in Chapter 6, *Handling Input*.

Setting the Colormap Segment

The first thing to note is that since the canvas pixwin is a region of the WIN_PIXWIN, you must also set the colormap segment for the canvas pixwin.

Color in Retained Canvases

If the canvas is retained, then the colormap segment must be set *before* CANVAS_RETAINED is set to TRUE. This is because the canvas package will determine the depth of the backing pixrect based on depth of the colormap segment defined for the WIN_PIXWIN. (If the colormap segment depth is greater than two, then the full depth of the display will be used. Otherwise, the backing pixrect depth will be set to one.)

Since the depth of the backing pixrect is determined when the canvas is created, you must create the canvas with CANVAS_RETAINED FALSE, then set the colormap segment, then set CANVAS_RETAINED to TRUE.

Color in Scrollable Canvases

If the canvas has scrollbars, you need to attach the scrollbars to the canvas *after* the colormap segment has been changed. If the canvas has already been created with scrollbars attached, you should change the colormap, then re-attach the scrollbars. This will insure that the scrollbar pixwin regions use the new colormap segment.

Example 5:

Below is an example of setting the colormap segment for a canvas:

```
#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <sunwindow/cms_rainbow.h>

init_color_canvas(base_frame)
Frame base_frame;
{
{
    Canvas          canvas;
    Pixwin          *pw;
    unsigned char    red[CMS_RAINBOWSIZE];
    unsigned char    green[CMS_RAINBOWSIZE];
    unsigned char    blue[CMS_RAINBOWSIZE];

    canvas = window_create(base_frame, CANVAS,
                           CANVAS_RETAINED, FALSE,
                           0);

    cms_rainbowsetup(red, green, blue);

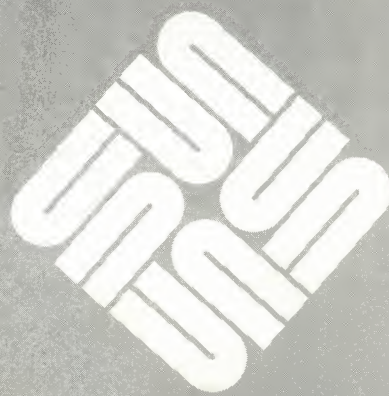
    /* set the WIN_PIXWIN colormap */
    pw = (Pixwin *) window_get(canvas, WIN_PIXWIN);
    pw_setcmsname(pw, CMS_RAINBOW);
    pw_putcolormap(pw, 0, CMS_RAINBOWSIZE, red, green, blue);

    /* set the CANVAS_PIXWIN colormap */
    pw = (Pixwin *) canvas_pixwin(canvas);
    pw_setcmsname(pw, CMS_RAINBOW);
    pw_putcolormap(pw, 0, CMS_RAINBOWSIZE, red, green, blue);

    window_set(canvas,
               CANVAS_RETAINED, TRUE,
               WIN_VERTICAL_SCROLLBAR, scrollbar_create(0),
               WIN_HORIZONTAL_SCROLLBAR, scrollbar_create(0),
               0);
}
}
```


Handling Input

Handling Input	77
6.1. An Overview of the Input Environment	79
How are events generated ?	79
What does the Notifier do with these events ?	80
How do windows determine which input they will receive?	80
6.2. Events	80
An event Procedure	80
How Subwindows Handle Events	81
6.3. A List of Events	81
Keyboard Motions	85
6.4. Classes of Events	86
ASCII Events	86
Locator Button Events	86
Locator Motion Events	86
Window Events	87
Function Key Events	88
Shift Key Events	89
Semantic Events	89
Other Events	89
6.5. Event Descriptors	90
6.6. Controlling Input in a Window	90
Input Focus	91
Input Mask	91



Determining which Window will Receive Input	92
6.7. Enabling and Disabling Events	93
Which Mask to Use	93
Setting the Input Mask as a Whole	95
Querying the Input Mask State	95
6.8. Querying and Setting the Event State	96
6.9. Releasing the Event Lock	97
6.10. Reading Events Explicitly	97

Handling Input

Material Covered

This chapter explains how input is handled in SunView. Specifically it:

- gives an overview on how input is handled in SunView
- describes *events* and how they are used;
- gives various classes of events —ASCII, action events, function keys, locator buttons, locator motion, window generated events, and so on;
- explains the input focus model distinguishing between *pick* and *keyboard* focuses;
- shows how to control where input is distributed using *input masks*;
- shows how to query the state of an event;
- shows how to explicitly read events.

The material in this chapter applies to the window system as a whole. However, it is of special interest to alerts or clients of canvases, who typically will want to handle events themselves.

Header Files

The definitions necessary to use SunView's input facilities are in the header file `<sunwindow/win_input.h>`, which is included by `<sunwindow/window_hs.h>`, which in turn is included by default when you include `<suntool/sunview.h>`.

Related Documentation

The chapter titled *Workstations* in the *SunView 1 System Programmer's Guide* explains the input system at a lower level, covering such topics as how to add user input devices to SunView.

Summary Listing and Tables

To give you a feeling for what you can do with events, a list of the available event descriptors and input related window events is given on the following page. Many of these are discussed in the rest of this chapter and elsewhere (use the *Index* to check). All are briefly described with their arguments in the input summary tables in Chapter 19, *SunView Interface Summary*:

- the *Event Descriptors* table begins on page 333;
- the *Input-Related Window Attributes* table begins on page 334.

Input-Related Window Attributes

WIN_INPUT_DESIGNEE	WIN_CONSUME_KBD_EVENTS
WIN_GRAB_ALL_INPUT	WIN_IGNORE_KBD_EVENTS
WIN_KBD_FOCUS	WIN_CONSUME_PICK_EVENT
WIN_KBD_INPUT_MASK	WIN_IGNORE_PICK_EVENT
WIN_PICK_INPUT_MASK	WIN_CONSUME_PICK_EVENTS
WIN_CONSUME_KBD_EVENT	WIN_IGNORE_PICK_EVENTS
WIN_IGNORE_KBD_EVENT	

Event Descriptors

WIN_NO_EVENTS	WIN_RIGHT_KEYS
WIN_ASCII_EVENTS	WIN_TOP_KEYS
WIN_IN_TRANSIT_EVENTS	WIN_UP_ASCII_EVENTS
WIN_LEFT_KEYS	WIN_UP_EVENTS
WIN_MOUSE_BUTTONS	

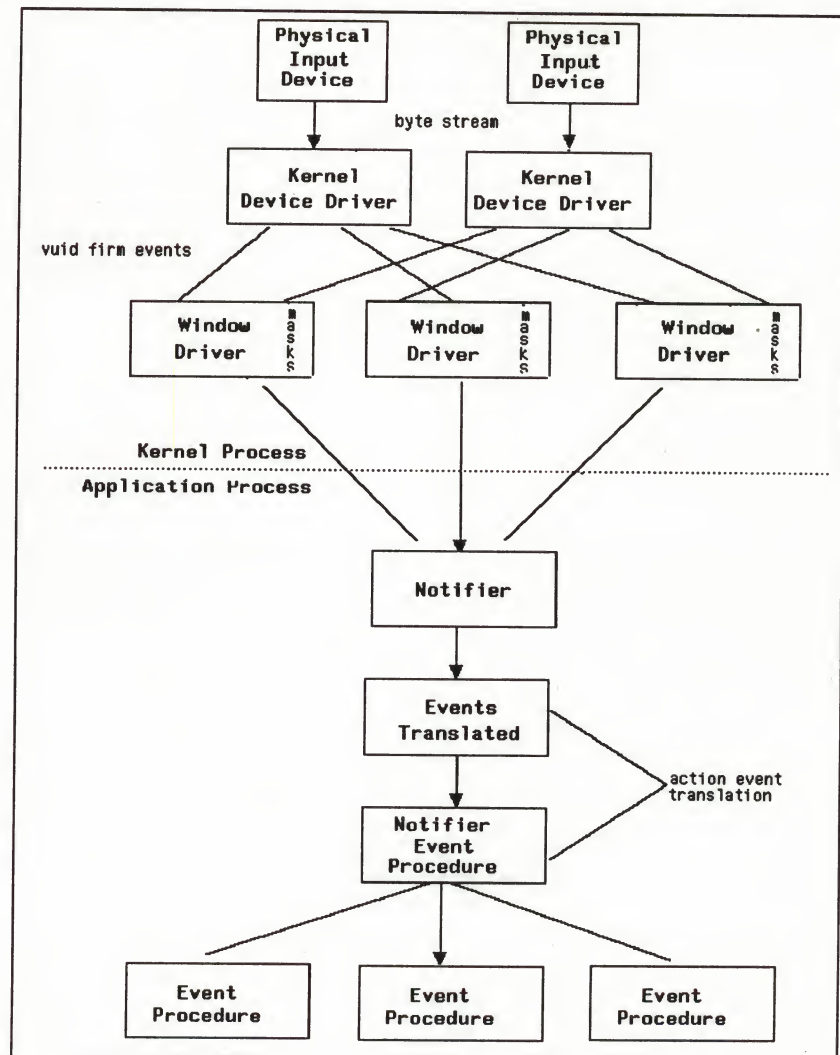
6.1. An Overview of the Input Environment

The input environment for SunView differs from UNIX programs. Most UNIX programs read characters from standard input by using either the `read(2)` system call or the standard I/O functions such as `getc(3S)`, `gets(3S)`, or `scanf(3S)`. SunView is different in that the underlying Notifier formats user input into uniform *events*, which it distributes to the window's *event procedure*.

How are events generated ?

Figure 6-1 illustrates how events are generated and handled in SunView.

Figure 6-1 *Input Events*



Events are generated from several sources. These include standard devices such as the keyboard and mouse, special input devices such as graphics tablets, and the window system itself.

SunView does not directly receive events from the hardware devices. Instead each user action is interpreted by a "virtual" user input device (VUID) interface. This interface packages the data it receives into an event and sends it to the application process.²⁷

What does the Notifier do with these events ?

The Notifier weaves events from all of these sources into a single, ordered *event stream*. This event stream eliminates the need for the application to poll separate streams from the different devices.

Because the underlying Notifier multiplexes the input stream between windows, each individual window operates under the illusion that it has the user's full attention. That is, it sees precisely those input events that the user has directed to it.

How do windows determine which input they will receive?

Each window indicates which events it is prepared to handle using *input masks*, described in Section 6.6, *Controlling Input in a Window*. These masks only let specified events through to the process.

6.2. Events

As discussed in the previous section, each user action generates an input event. This event is passed to your event procedure as an Event pointer (type `Event *`). Three types of information are encoded as part of an event:

- an identifying code, accessed with the macro `event_action()`
- the location of the event in the window's coordinate system, accessed with the macros `event_x()` and `event_y()`
- a timestamp, accessed with the macro `event_time()`

Notice that the macro `event_action()` has replaced the old `event_id()`. For compatibility reasons, `event_id()` is still supported, so that old code that does not use the new action event codes will still work. See Section 6.4, *Classes of Events*, for an explanation of action events. New programs that want to take advantage of the new action events must use the `event_action()` macro.

An event Procedure

Use the following form to specify an event procedure in your applications:

```
void
sample_event_proc(window, event, arg)
    Window    window;
    Event      *event;
    caddr_t    arg;
```

²⁷ It is possible to bypass the VUID and receive unencoded events. Refer to the section on *Unencoded Input* in Chapter 7 of the *SunView 1 System Programmer's Guide*.

How Subwindows Handle Events

The arguments passed in are the window, the event, and an optional argument containing data pertaining to the event. For example, if the event is a `SCROLL_REQUEST`, `arg` will be the scrollbar that sent the event.

The canvas and panel subwindows pass events that they receive on to an event procedure. These event procedures are supplied by the application as the value of `WIN_EVENT_PROC`. If you set the `WIN_EVENT_PROC` of a canvas or panel to a function you have written, you can receive events after they have been processed by the canvas or panel. Both the canvas and panel packages process `SCROLL_REQUEST`, `WIN_RESIZE`, and `WIN_REPAINT` events before calling your event procedure. The form of an event procedure is:

```
void
sample_event_proc(window, event, arg)
    Window      window;
    Event        event;
    caddr_t      arg;
```

The arguments passed in are the window (canvas or panel), the event, and an optional argument containing data pertaining to the event. For example, if the event is a `SCROLL_REQUEST`, `arg` will be the scrollbar that sent the event.

The default panel event procedure maps events to actions and determines which panel item to send the event to. The default canvas event procedure does no further processing of the event. You can call the default window event procedure by calling `window_default_event_proc()` with the same arguments passed to your event procedure.²⁸

6.3. A List of Events

Two tables are given on the following pages. Table 6-1, *Event Codes*, lists the predefined event codes and their values.²⁹ The event id or code numbers that the window system uses to represent an event are included in this table. These event code numbers are in the range of 0-65535. The numbers are useful when debugging a program because the debugger reports event codes as decimal integers and not as names.

Table 6-2, *Keyboard Motions and Accelerators*, lists the event name and its associated keyboard accelerator.

²⁸ If you need to receive an event before it is processed by a canvas, panel, or any other type of window, you can use the more general notifier interposition mechanism described in Chapter 17, *The Notifier*,

²⁹ The same table also appears in the input summary section of Chapter 19, *SunView Interface Summary*.

Table 6-1 *Event Codes*

<i>Event Code</i>	<i>Description</i>	<i>Value (for debugging)</i>
ASCII_FIRST	Marks beginning of ASCII range	0
ASCII_LAST	Marks end of ASCII range	127
META_FIRST	Marks beginning of META range	128
META_LAST	Marks end of META range	255
ACTION_ERASE_CHAR_BACKWARD	Erase char to the left of caret	31744
ACTION_ERASE_CHAR_FORWARD	Erase char to the right of caret	31745
ACTION_ERASE_WORD_BACKWARD	Erase word to the left of caret	31746
ACTION_ERASE_WORD_FORWARD	Erase word to the right of caret	31747
ACTION_ERASE_LINE_BACKWARD	Erase to the beginning of the line	31748
ACTION_ERASE_LINE_END	Erase to the end of the line	31749
ACTION_GO_CHAR_BACKWARD	Move the caret one character to the left	31752
ACTION_GO_CHAR_FORWARD	Move the caret one character to the right	31753
ACTION_GO_WORD_BACKWARD	Move the caret one word to the left	31754
ACTION_GO_WORD_END	Move the caret to the end of the word	31756
ACTION_GO_WORD_FORWARD	Move the caret one word to the right	31755
ACTION_GO_LINE_BACKWARD	Move the caret to the start of the line	31757
ACTION_GO_LINE_END	Move the caret to the end of the line	31759
ACTION_GO_LINE_FORWARD	Move the caret to the start of the next line	31758
ACTION_GO_COLUMN_BACKWARD	Move the caret up one line, maintaining column position	31761
ACTION_GO_COLUMN_FORWARD	Move the caret down one line, maintaining column position	31762
ACTION_GO_DOCUMENT_START	Move the caret to the beginning of the text	31763
ACTION_GO_DOCUMENT_END	Move the caret to the end of the text	31764
ACTION_STOP	Stop the operation	31767
ACTION_AGAIN	Repeat previous operation	31768
ACTION_PROPS	Show property sheet window	31769
ACTION_UNDO	Undo previous operation	31770
ACTION_FRONT	Bring window to the front of the desktop	31772
ACTION_BACK	Put the window at the back of the desktop	31773
ACTION_OPEN	Open a window from its icon form or close if already open)	31775
ACTION_CLOSE	Close a window to an icon	31776
ACTION_COPY	Copy the selection to the clipboard	31774
ACTION_PASTE	Copy clipboard contents to the insertion point	31777
ACTION_CUT	Delete the selection, put on clipboard	31781
ACTION_COPY_THEN_PASTE	Copies then pastes text	31784
ACTION_FIND_FORWARD	Find the text selection to the right of the caret	31779
ACTION_FIND_BACKWARD	Find the text selection to the left of the caret	31778
ACTION_FIND_AND_REPLACE	Show find and replace window	31780
ACTION_SELECT_FIELD_FORWARD	Select the next delimited field	31783
ACTION_SELECT_FIELD_BACKWARD	Select the previous delimited field	31782

Table 6-1 *Event Codes—Continued*

<i>Event Code</i>	<i>Description</i>	<i>Value (for debugging)</i>
ACTION_MATCH_DELIMITER	Selects text up to a matching delimiter	31894
ACTION_QUOTE	Causes next event in the input stream to pass untranslated by the keymapping system	31898
ACTION_EMPTY	Causes the subwindow to be emptied	31899
ACTION_STORE	Stores the specified selection as a new file	31785
ACTION_LOAD	Loads the specified selection as a new file	31786
ACTION_GET_FILENAME	Gets the selected filename	31788
ACTION_SET_DIRECTORY	Sets the directory to the selection	31788
ACTION_INCLUDE_FILE	Selects the current line (in pending-delete mode) and attempts to insert the file described by that selection	31891
ACTION_CAPS_LOCK	Toggle caps lock state	31895
PANEL_EVENT_CANCEL	The panel or panel item is no longer "current"	32000
PANEL_EVENT_MOVE_IN	The panel or panel item was entered with no mouse buttons down	32001
PANEL_EVENT_DRAG_IN	The panel or panel item was entered with one or more mouse buttons down	32002
SCROLL_REQUEST	Scrolling has been requested	32256
SCROLL_ENTER	Locator (mouse) has moved into the scrollbar	32257
SCROLL_EXIT	Locator (mouse) has moved out of the scrollbar	32258
LOC_MOVE	Locator (mouse) has moved	32512
LOC_STILL	Locator (mouse) has been still for 1/5 second	32513
LOC_WINENTER	Locator (mouse) has entered window	32514
LOC_WINEXIT	Locator (mouse) has exited window	32515
LOC_DRAG	Locator (mouse) has moved while a button was down	32516
LOC_RGNENTER	Locator (mouse) has entered a region of the window	32519
LOC_RGNEXIT	Locator (mouse) has exited a region of the window	32520
LOC_TRAJECTORY	Inhibits the collapse of mouse motions; clients receive LOC_TRAJECTORY events for every locator motion the window system detects.	32523
WIN_REPAINT	Some portion of window requires repainting	32517
WIN_RESIZE	Window has been resized	32518
WIN_STOP	User has pressed the <i>stop</i> key	32522
KBD_REQUEST	Window is about to become the focus of keyboard input	32526
KBD_USE	Window is now the focus of keyboard input	32524
KBD_DONE	Window is no longer the focus of keyboard input	32525
SHIFT_LEFT	Left shift key changed state	32530
SHIFT_RIGHT	Right shift key changed state	32531
SHIFT_CTRL	Control key changed state	32532
SHIFT_META	Meta key changed state	32534
SHIFT_LOCK	Shift lock key changed state	32529
SHIFT_CAPSLOCK	Caps lock key changed state	32528

Table 6-1 *Event Codes— Continued*

<i>Event Code</i>	<i>Description</i>	<i>Value (for debugging)</i>
BUT (i)	Locator (mouse) buttons 1–10	BUT(1) is 32544
MS_LEFT	Left mouse button	32544
MS_MIDDLE	Middle mouse button	32545
MS_RIGHT	Right mouse button	32546
KEY_LEFT (i)	Left function keys 1–15	KEY_LEFT (1) is 32554
KEY_RIGHT (i)	Right function keys 1–15	KEY_RIGHT (1) is 32570
KEY_TOP (i)	Top function keys 1–15	KEY_TOP (1) is 32586

Table 6-2 Keyboard Motions and Accelerators

Command Token	SunView 4.0	SunView 3.x
ACTION_ERASE_CHAR_BACKWARD	Delete	Delete
ACTION_ERASE_CHAR_FORWARD	Shift-Delete	Shift-Delete
ACTION_ERASE_WORD_BACKWARD	Control-W	Control-W
ACTION_ERASE_WORD_FORWARD	Shift-Control-W	Shift-Control-W
ACTION_ERASE_LINE_BACKWARD	Control-U	Control-U
ACTION_ERASE_LINE_END	Shift-Control-U	Shift-Control-U
ACTION_GO_CHAR_BACKWARD	Control-B or Shift-Control-F or R10	
ACTION_GO_CHAR_FORWARD	Control-F or Shift-Control-B or R12	
ACTION_GO_WORD_BACKWARD	Control-comma or Shift-Control-period or Shift-Control-slash	
ACTION_GO_WORD_END	Control-period	
ACTION_GO_WORD_FORWARD	Control-slash or Shift-Control-comma	
ACTION_GO_LINE_FORWARD	Control-semicolon or R11	
ACTION_GO_LINE_BACKWARD	Control-A or Shift-Control-E	
ACTION_GO_LINE_END	Control-E or Shift-Control-A	
ACTION_GO_COLUMN_BACKWARD	Control-P or Shift-Control-N or R14	
ACTION_GO_COLUMN_FORWARD	Control-N or Shift-Control-P or R18	
ACTION_GO_DOCUMENT_START	Shift-Control-Return or R7	
ACTION_GO_DOCUMENT_END	Control-Return or R13	Control-Return
ACTION_STOP	L1	L1
ACTION_AGAIN	L2 or Meta-A	L2
ACTION_PROPS	L3	L3
ACTION_UNDO	L4 or Meta-U	L4
ACTION_FRONT	L5	L5
ACTION_BACK	Shift-L5	Shift-L5
ACTION_OPEN	L7	L7
ACTION_CLOSE	Shift-L7	Shift-L7
ACTION_COPY	L6 or Meta-C	L6
ACTION_PASTE	L8 or Meta-V	L8 or Control-G
ACTION_CUT	L10 or Meta-X	L10 or Control-D
ACTION_COPY_THEN_PASTE	Meta-P	Control-P
ACTION_FIND_FORWARD	L9 or Meta-F	L9 or Control-F
ACTION_FIND_BACKWARD	Shift-L9 or Shift-Meta-F	Shift-L9 or Shift-Control-F
ACTION_FIND_AND_REPLACE	Control-L9	
ACTION_SELECT_FIELD_FORWARD	Control-Tab	
ACTION_SELECT_FIELD_BACKWARD	Shift-Control-Tab	
ACTION_MATCH_DELIMITER	Meta-D	
ACTION_QUOTE	Meta-O	
ACTION_EMPTY (Document)	Meta-E	
ACTION_STORE	Meta-S	

Table 6-2 Keyboard Motions and Accelerators—Continued

Command Token	SunView 4.0	SunView 3.x
ACTION_LOAD	Meta-L	
ACTION_INCLUDE_FILE	Meta-I	
ACTION_HELP ³⁰	Meta-? (Meta-Shift-/)	
ACTION_GET_FILENAME	Escape	Escape
ACTION_CAPS_LOCK	T1	T1

6.4. Classes of Events

This section groups each of the events described in Table 6-1, *Event Codes*, into logical classes. Each class is described below.

ASCII Events

The event codes in the range 0 to 255 inclusive are assigned to the ASCII event class. This includes the standard 7-bit ASCII codes and their 8-bit META counterparts.

If a user strikes a key which has an obvious ASCII meaning; that is, a key in the main typing array labeled with a single letter, it causes the VUID to enqueue for the appropriate window an event whose code is the corresponding 7-bit ASCII character.

The *META* event code values (128 through 255) are generated when the user strikes a key that would generate a 7-bit ASCII code while the META key is also depressed.

Locator Button Events

The standard Sun locator is a three button mouse, whose buttons generate the event codes MS_LEFT, MS_MIDDLE and MS_RIGHT.

In general, a physical locator can have up to 10 buttons connected to it. In some cases, the locator itself may not have any buttons on it; however, it may have buttons from another device assigned to it. A light pen is an example of such a locator.

Each button that is associated with the VUID's locator is assigned an event code; the *i*-th button is assigned the code BUT (i). Thus the event codes MS_LEFT, MS_MIDDLE and MS_RIGHT correspond to BUT (1), BUT (2) and BUT (3).

Locator Motion Events

The physical locator constantly provides an (x, y) coordinate position in pixels; this position is transformed by SunView to the coordinate system of the window receiving an event. Locator motion event codes include LOC_MOVE, LOC_DRAG, LOC_TRAJECTORY, and LOC_STILL.

Since the locator tracking mechanism reports the current position at a set sampling rate, 40 times per second, fast motions will yield non-adjacent locations in consecutive events.

³⁰ If your keyboard has the **L16** key, you may also use it.

A `LOC_MOVE` event is reported when the locator moves, regardless of the state of the locator buttons. If you only want to know about locator motion when a button is down, then enable `LOC_DRAG` instead of `LOC_MOVE`. This will greatly reduce the number of motion events that your application has to process.

When you enable `LOC_MOVE` or `LOC_DRAG`, the window system gives you the current locator position by collapsing consecutive locator motion events into one. This operation is appropriate for applications such as dragging an image from one point to another, in which it is important to keep up with the mouse cursor.

For some applications, however, each point on the cursor trajectory is of interest; for example, a program that lets the user draw. In these situations you may not want to collapse consecutive motion events. In such a situation you should ask for `LOC_TRAJECTORY` events, which suppresses any event collapsing so that you get all the locator movements that the window system sees.

Note that when you ask for `LOC_TRAJECTORY` events, you get (many!) `LOC_TRAJECTORY` events in place of `LOC_MOVE`'s, but you still get `LOC_DRAG` events if you have enabled them.

If you ask for `LOC_STILL`, a single `LOC_STILL` event will be reported after the locator has been still for 1/5 of a second.

Window Events

Window events are generated by the window system itself. They are meaningful only to the window to which they are directed.

To be informed when the locator enters or exits a window, enable events with the codes `LOC_WINENTER` and `LOC_WINEXIT`.

NOTE *If you are using the tile mechanism described in the *SunView 1 System Programmer's Guide*, then you will be told when the locator has entered or exited a tile using the `LOC_RGNENTER` and `LOC_RGNEXIT` events. To receive these events you must also have `LOC_MOVE` enabled.*

Resize & Repaint Events

When the size of a window is changed (either by the user or programmatically) a `WIN_RESIZE` event is generated to give the client a chance to adjust any relevant internal state to the new window size. You should *not* repaint the screen on receiving a resize event. You will receive a separate `WIN_REPAINT` event when a portion of the window needs to be repainted.

NOTE *If you are using a canvas subwindow you will not need to track resize and repaint events directly. The canvas package receives these events, computes the new window dimensions or the precise area requiring repainting, and calls your resize or repaint procedures directly. See Chapter 5, *Canvases* for more details.*

Keyboard Focus Events

Three events let your application interact with the keyboard focus mechanism (the keyboard focus is explained in section 6.6, *Controlling Input in a Window*). When the user explicitly directs the keyboard focus towards your window, you will receive a `KBD_REQUEST` event. Your window will then become the keyboard focus unless you call `window_refuse_kbd_focus()`. Refusing the keyboard focus, when you don't need it, contributes to the usefulness of the split keyboard/pick focus mode available as a runtime option to *sunview(1)*.

The events `KBD_USE` and `KBD_DONE` parallel the locator events `LOC_WINENTER` and `LOC_WINEXIT`, respectively. `KBD_USE` indicates that your window now has the keyboard focus and `KBD_DONE` indicates that your window no longer has it.

Stop Event

If the user presses and releases the **Stop** key, an event with the code `WIN_STOP` will be sent to the window under the cursor.³¹ In addition, a `SIGURG` signal is sent to the window's process. Your application can use the **Stop** key by clearing a stop flag and setting a `SIGURG` interrupt handler³² before entering a section of code that might, from the user's perspective, take a long time. If your `SIGURG` handler is called, set the stop flag and return. In the code that is taking a long time, query the stop flag whenever convenient. When you notice that the stop flag has been set, read the event, then gracefully terminate your long operation.

Function Key Events

The function keys in the `VUID` define an idealized standard layout that groups keys by location: 15 left, 15 right, 15 top and 2 bottom.³³

The event codes associated with the function keys are `KEY_LEFT(i)`, `KEY_RIGHT(i)` and `KEY_TOP(i)`, where *i* ranges from 1 to 15.

If you specifically ask for a function key event code, then that event code will be passed to your event procedure.

If you *don't* specifically ask for a given function key event code, then when the user presses that function key you will get an escape sequence instead of the function key event code (assuming ASCII events have been enabled). For physical keystations that are mapped to cursor control keys, events with codes that correspond to the ANSI X3.64 7-bit ASCII encoding for the cursor control function are transmitted. For physical keystations mapped to other function keys, events with codes that correspond to an ANSI X3.64 user-definable escape sequence are transmitted.

³¹ `WIN_STOP` only works when enabled in the `PICK` event mask and not in the `KBD` event mask.

³² See `notify_set_signal_func()` in Chapter 17, *The Notifier*

³³ The actual position of the function keys on a given physical keyboard may differ — see `kbd(5)` for details on various keyboards.

Shift Key Events

Applications can notice when a shift key changes state by enabling events with the following codes: `SHIFT_LEFT`, `SHIFT_RIGHT`, `SHIFT_CTRL`, `SHIFT_META`, `SHIFT_LOCK` and `SHIFT_CAPSLOCK`. Although these codes allow you to treat one or more shift key as function-invoking keys, this is not recommended. Instead of watching for the event directly, you should query the state of the shift keys via the macros described on the next page.

Semantic Events

Release 4.0 of the SunOS introduces a new type of event. These events are called action events and represent some old and many new functions in the window system. They are similar to the old events in that they are mapped to specific keys on the keyboard. That is, certain combinations of keystrokes in SunView correspond to high-level action events. For example, pressing the `[Copy]` key copies the current selection to the Clipboard in text subwindows, panels and tty subwindows.

Action events differ from the old events in that applications can directly express interest in the high-level action, "Copy the selection to the Clipboard" rather than in the low-level, "The L6 key was pushed". These events appear in Table 6-1 with the prefix `ACTION_`. Applications *should* use action events, because left-handed users can assign `[Copy]` to a different key, and in the future users will be allowed to tie high-level events to arbitrary key combinations.

Other Events

Your application may receive events which don't fall into any of the classes described above. For example, a non-standard input device, such as a second mouse, may emit its own types of events. Also, a software object may communicate with other software objects via events, as is the case when a scrollbar sends a `SCROLL_REQUEST` to a panel or a canvas.

In general, your event procedure should not treat such unexpected events as errors. They can simply be ignored.

6.5. Event Descriptors

Events have been further grouped into descriptors. Descriptors describe classes of events such as all ASCII events, all mouse buttons, all top function keys, and so on. You will use these descriptors to set input masks, described in Section 6.7 *Enabling and Disabling Events*

The descriptors are summarized in the following table.

Table 6-3 *Event Descriptors*

<i>Event Descriptor</i>	<i>Explanation</i>
WIN_NO_EVENTS	Clears input mask — no events will be accepted. Note: the effect is the same whether used with a <i>consume</i> or an <i>ignore</i> attribute. A new window has a cleared input mask.
WIN_ASCII_EVENTS	All ASCII events. ASCII events that occur while the META key is depressed are reported with codes in the META range. In addition, cursor control keys and function keys are reported as ANSI escape sequences: a sequence of events whose codes are ASCII characters, beginning with <ESC>.
WIN_IN_TRANSIT_EVENTS LOC_WINENTER, and	Enables immediate LOC_MOVE, LOC_WINEXIT events. Pick mask only. Off by default.
WIN_LEFT_KEYS	The left function keys, KEY_LEFT(1) — KEY_LEFT(15).
WIN_MOUSE_BUTTONS MS_MIDDLE and MS_LEFT.	Shorthand for MS_RIGHT, Also sets or resets WIN_UP_EVENTS.
WIN_RIGHT_KEYS	The right function keys, KEY_RIGHT(1) — KEY_RIGHT(15).
WIN_TOP_KEYS	The top function keys, KEY_TOP(1) — KEY_TOP(15).
WIN_UP_ASCII_EVENTS	Causes the matching up transitions to normal ASCII events to be reported — if you see an 'a' go down, you'll eventually see the matching 'a' up.
WIN_UP_EVENTS	Causes up transitions to be reported for button and function key events being consumed.

6.6. Controlling Input in a Window

Input may be controlled using *input focus* and *input mask*. The input focus is the window that is currently receiving input. The input mask specifies which events a window will receive and which events a window will ignore. This section introduces these concepts and gives the algorithm used by the window system to decide which window will receive a given input.

Input Focus

SunView supports two types of focus models, a single focus model and a split focus model.

The single focus model specifies that all input, no matter which input device it came from, goes to the same window. The split input focus lets the user control the *pick input focus* and the *keyboard input focus* separately.

The word *pick* comes from the general graphics term *pick device*, which is a user input device that allows you to move a cursor on the screen and then click a button to choose a point on the screen. The most common pick devices are the mouse, light pen and graphics tablet.

Under the split input focus model, mouse clicks and keystrokes may be distributed to different windows. This makes some operations easier for the user. For example, the user can select text in one window and move it to another window without having to position the cursor over the destination window.

In general, the user controls the keyboard focus by using specific button clicks and controls the pick focus by moving the mouse. Sometimes, it is appropriate for input focuses to be under program control. Generally you should only change an input focus based on some explicit and predictable user action.

You can indicate that you want a window to become the keyboard focus by setting the `WIN_KEYBOARD_FOCUS` attribute to `TRUE`. Note that this is only a hint to the window system. If the keyboard focus is tied to the pick focus, then this call has no effect. The target window might also refuse the keyboard focus request generated by this call (see `KBD_REQUEST` under *Window Events* above). You can set the pick focus via the `WIN_MOUSE_XY` attribute, which sets the mouse cursor to a particular position within a window.

For example, the call

```
window_set(win, WIN_MOUSE_XY, 200, 300, 0);
```

sets the cursor to the window-relative position (200, 300) and sets the pick focus to win.

Input Mask

An input mask specifies which events a window will receive and which events it will ignore. In other words, an input mask serves as a read enable mask. Each window has both a *pick input mask*, to specify which pick related events it wants, and a *keyboard input mask*, to specify which keyboard related events it wants.

When a window is the pick focus, its pick mask is used to screen events. When a window is the keyboard focus, its keyboard mask is used to screen events.

This section describes how to specify which events a window will receive and which it will ignore.

Determining which Window will Receive Input

The Notifier determines which window will receive a given event according to the following algorithm:

- First, the keyboard input mask for the window which is the keyboard focus is checked to see if it wants the event. If so, then it becomes the recipient; otherwise the next test is applied.
- Second, the pick input mask for the window which is under the cursor is checked to see if it wants the event. If several windows are layered under the cursor, then the event is tested against the pick input mask of the topmost window. If the mask wants the event, then it becomes the recipient; otherwise the next test is applied.
- If the event does not match the pick input mask of the window under the cursor, then the event will be offered to that window's *designee*. By default the *designee* is the window's owner. You can set the designee explicitly by calling `window_set()` with the `WIN_INPUT_DESIGNEE` attribute.³⁴
- If an event is offered unsuccessfully to the root window, it is discarded. Windows which are not in the chain of designated recipients never have a chance to accept the event.
- Occasionally you may want to specify that a given window is to receive *all* events, regardless of their location on the screen. You can do this by setting the `WIN_GRAB_ALL_INPUT` attribute for the window to `TRUE`.
- If a recipient is found, then the locator coordinates are adjusted to the coordinate system of the recipient, and the event is appended to the recipient's input stream. Thus, every window sees a single ordered stream of time-stamped input events, which contain only the events that a window has declared to be of interest.

³⁴ Note that you must give the `WIN_DEVICE_NUMBER` of the window you wish to be the designee, not its handle. This is to allow specifying windows in another user process as the input designee. So the following call would set `win2` to be the designee for `win1`: `window_set(win1, WIN_INPUT_DESIGNEE, window_get(win2, WIN_DEVICE_NUMBER));`

6.7. Enabling and Disabling Events

You specify which events a window will receive and which it will ignore by setting the window's input masks via the following set of attributes:

Table 6-4 *Attributes Used to Set Window Input Masks*

<i>Events Taking a Single Code</i>	<i>Events Taking a Null Terminated List</i>
WIN_CONSUME_KBD_EVENT	WIN_CONSUME_KBD_EVENTS
WIN_IGNORE_KBD_EVENT	WIN_IGNORE_KBD_EVENTS
WIN_CONSUME_PICK_EVENT	WIN_CONSUME_PICK_EVENTS
WIN_IGNORE_PICK_EVENT	WIN_IGNORE_PICK_EVENTS

The above attributes take as values either event codes such as LOC_MOVE, MS_LEFT, KEY_LEFT(2), and so on, or *event descriptors*. The attributes in the left column, ending in “_EVENT”, take a single code or descriptor, while those on the right, ending in “_EVENTS”, take a null terminated list.

Which Mask to Use

To enable or disable ASCII events, use the keyboard mask. To enable or disable locator motion and button events, use the pick mask.

Function keys are typically associated with the keyboard mask, but sometimes it makes sense to include some function keys in the pick mask — in effect extending the number of buttons associated with the pick device. For example, in the SunView interface the **Again**, **Undo**, **Copy**, **Paste**, **Cut**, and **Find** function keys are associated with the keyboard mask, while the **Stop**, **Front**, and, **Open** keys are associated with the pick mask.

Examples

The event attributes cause precisely the events you specify to be enabled or disabled — the input mask is *not* automatically cleared to an initial state. To be sure that an input mask will let through the events you specify, first clear the mask with the special WIN_NO_EVENTS descriptor. Take, for example, the following two calls:

```

window_set(win, WIN_CONSUME_PICK_EVENTS,
            WIN_MOUSE_BUTTONS, LOC_DRAG, 0,
            0);

window_set(win, WIN_CONSUME_PICK_EVENTS,
            WIN_NO_EVENTS, WIN_MOUSE_BUTTONS, LOC_DRAG, 0,
            0);

```

The first call adds the mouse buttons and LOC_DRAG to the existing pick input mask, while the second call sets the mask to let *only* the mouse buttons and LOC_DRAG through.

Canvases by default enable LOC_WINENTER, LOC_WINEXIT, LOC_MOVE, and the three mouse buttons, MS_LEFT, MS_MIDDLE, and MS_RIGHT.³⁵ You could allow the user to type in text to a canvas by calling:

```
window_set(canvas, WIN_CONSUME_KBD_EVENT, WIN_ASCII_EVENTS, 0);
```

Sometime later you could disable type-in by calling:

```
window_set(canvas, WIN_IGNORE_KBD_EVENT, WIN_ASCII_EVENTS, 0);
```

An application needing to track mouse motion with the button down would enable LOC_DRAG by calling:

```
window_set(canvas, WIN_CONSUME_PICK_EVENT, LOC_DRAG, 0);
```

You can enable or disable the left, right or top function keys as a group via the event descriptors WIN_LEFT_KEYS, WIN_RIGHT_KEYS, or WIN_TOP_KEYS. Note that if you want to see the up event you must also ask for WIN_UP_EVENTS, as in:

```
window_set(win, WIN_CONSUME_KBD_EVENTS, WIN_LEFT_KEYS,  
            WIN_UP_EVENTS, 0);
```

In order to improve interactive performance, in the default case, windows do not receive locator motion events (LOC_WINENTER, LOC_WINEXIT, and LOC_MOVE) until after a LOC_STILL has been generated. If each window responds to all of the events that are generated each time the mouse passes over the window, then the response time of the system will be slowed down. Each window will "wake up" when the mouse passes over it on the way to somewhere else on the screen.

If you want a window to receive all events, even if the mouse is just passing over the window without stopping, enable WIN_IN_TRANSIT_EVENTS, with a call such as:

```
window_set(canvas, WIN_CONSUME_PICK_EVENTS,  
            WIN_IN_TRANSIT_EVENTS, 0);
```

³⁵ Note that the canvas package expects to receive these events, and will not function properly if you disable them.

Setting the Input Mask as a Whole

The attributes `WIN_KBD_INPUT_MASK` and `WIN_PICK_INPUT_MASK` allow you to get or set an entire input mask. Let's take the example of a subroutine that provides interactive feedback. You can save the input mask on entry to the subroutine, set up the mask as appropriate, and restore the original mask before returning as follows:

```
do_feedback()
{
    Inputmask *saved_mask;

    saved_mask = (Inputmask *)
        window_get(win, WIN_KBD_INPUT_MASK);
    ...
    window_set(win, WIN_KBD_INPUT_MASK, saved_mask, 0);
}
```

Keep in mind that the `inputmask` pointer returned by `window_get()` points to a static structure which is shared by all windows in the application. Getting either the keyboard or pick input masks for another window will cause the static structure to be overwritten.

Querying the Input Mask State

You can use `window_get()` with `WIN_CONSUME_PICK_EVENT` and `WIN_CONSUME_KBD_EVENT` to query the state of the input masks. For example, the following call will find out whether or not a canvas is accepting `LOC_DRAGs`:

```
flag = (int)window_get(canvas, WIN_CONSUME_PICK_EVENT, LOC_DRAG);
```

6.8. Querying and Setting the Event State

You can query the state associated with an event using the following macros, all of which take as their only argument a pointer to an Event.

Table 6-5 *Macros to Get the Event State*

Macro	Returns
<code>event_action()</code>	The identifying code of the event. The codes are discussed in the previous section. ³⁶
<code>event_is_up()</code>	TRUE if the event is a button or key event and the state is up.
<code>event_is_down()</code>	TRUE if the event is a button or key event and the state is down.
<code>event_x()</code>	The x coordinate of the locator in the window's coordinate system at the time the event occurred.
<code>event_y()</code>	The y coordinate of the locator in the window's coordinate system at the time the event occurred.
<code>event_shiftmask()</code>	The value of predefined shift-keys (described in <code>kbd(5)</code>). Possible values: <pre>#define CAPSMASK 0x0001 #define SHIFTMASK 0x000E #define CTRLMASK 0x0030 #define META_SHIFT_MASK 0x0040</pre>
<code>event_time()</code>	The event's timestamp, formatted as a <code>timeval</code> struct, as defined in <code><sys/time.h></code> .
<code>event_shift_is_down()</code>	TRUE if one of the shift keys are down.
<code>event_ctrl_is_down()</code>	TRUE if the control key is down.
<code>event_meta_is_down()</code>	TRUE if the meta key is down.
<code>event_is_button()</code>	TRUE if the event is a mouse button.
<code>event_is_ascii()</code>	TRUE if the event is in the ASCII range (0 thru 127).
<code>event_is_meta()</code>	TRUE if the event is in the META range (128 thru 255).
<code>event_is_key_left()</code>	TRUE if the event is any <code>KEY_LEFT(i)</code> .
<code>event_is_key_right()</code>	TRUE if the event is any <code>KEY_RIGHT(i)</code> .
<code>event_is_key_top()</code>	TRUE if the event is any <code>KEY_TOP(i)</code> .

In addition to the above macros, which tell about the state of a particular event, you can query the state of any button or key via the `WIN_EVENT_STATE` attribute. For example, to find out whether or not the first right function key is down you would call:

```
kl_down = (int)
          window_get(canvas, WIN_EVENT_STATE, KEY_RIGHT(1));
```

The call will return non-zero if the key is down, and zero if the key is up.

The following macros are provided to let you set some of the states associated with an event.

³⁶ `event_id()` is replaced by `event_action()`. However, for compatibility, `event_id()` will still be supported.

Table 6-6 *Macros to Set the Event State*

<i>Macro</i>	<i>Effect</i>
<code>event_set_action(event, code)</code>	set event's id to code.
<code>event_set_shiftmask</code> <code>(event, shiftmask)</code>	set event's shiftmask to shiftmask. Possible values: #define CAPSMASK 0x0001 #define SHIFTMASK 0x000E #define CTRLMASK 0x0030 #define META_SHIFT_MASK 0x0040
<code>event_set_x(event, x)</code>	set event's x coordinate to x.
<code>event_set_y(event, y)</code>	set event's y coordinate to y.
<code>event_set_time(event, time)</code>	set event's timestamp to time.
<code>event_set_up(event)</code>	set state of a button event to up.
<code>event_set_down(event)</code>	set state of a button event to down.

6.9. Releasing the Event Lock

If an operation generated by an input event is going to take over 5 seconds, then call this routine to allow other processes to get input:³⁷

```
void
window_release_event_lock(window)
    Window window;
```

6.10. Reading Events Explicitly

There are times when it is appropriate to go get the next event yourself, rather than waiting for it to come through the normal event stream from the Notifier. In particular, when tracking the mouse with an image which requires significant computation, it may be desirable to read events until a particular action, such as a mouse button up, is detected. To read the next input event for a window, bypassing the Notifier, use the function:

```
int
window_read_event(window, event)
    Window window;
    Event *event;
```

`window_read_event()` fills in the event structure, and returns 0 if all went well. In case of error, it sets the global variable `errno` and returns -1.

`window_read_event()` can be used in either a blocking or non-blocking mode, depending on how the window has been set up.³⁸

³⁷ For more details see the section on synchronization in the *Workstations* chapter of the *SunView 1 System Programmer's Guide*.

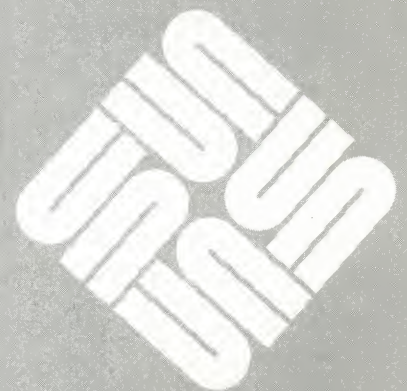
³⁸ `window_read_event()` is the high-level library standard function equivalent of `input_readevent()` in the low-level library. For further information, see Section 5.6, *Reading Input* in the *SunView 1 System Programmer's Guide*.

Note that if you read events in a canvas subwindow yourself, you must translate the event's location to canvas space by calling `canvas_event()`:

```
event_in_canvas_space = canvas_event(canvas, event);
```

Imaging Facilities: Pixwins

Imaging Facilities: Pixwins	101
7.1. What is a Pixwin?	103
7.2. Accessing a Pixwin's Pixels	103
Obtaining the Window's Pixwin	103
Write Routines	104
Basic RasterOp Operations	104
Other Raster Operations	104
Text Routines	105
Batching and Stenciling Routines	106
Drawing Polygons	107
Drawing Curved Shapes	107
Drawing Lines	107
Read and Copy Routines	108
7.3. Rendering Speed	108
Locking	109
Batching	110
Locking and Batching Interaction	112
7.4. Clipping With Regions	112
7.5. Color	113
Introduction to Color	113
The Colormap	113
Changing the Colormap	114
Colormap Segments	114



Background and Foreground	115
Default Colormap Segment	115
Changing Colors from the Command Line	115
Sharing Colormap Segments	115
Example: <i>showcolor</i>	116
Manipulating the Colormap	117
Cycling the Colormap	118
Miscellaneous Utilities	118
Using Color	119
Cursors and Menus	119
Is My Application Running on a Color Display?	119
Simulating Grayscale on a Color Display	120
Software Double Buffering	120
Hardware Double-Buffering	122
7.6. Plane Groups and the <i>cgfour</i> Frame Buffer	124
SunView and Plane Groups	125
sunview and Plane Groups	126

Imaging Facilities: Pixwins

Material Covered

This chapter describes the *pixwin* which is the construct you use to draw or render images in SunView. The most basic use of pixwins is to draw in a canvas subwindow.

In addition to basic pixwin usage, this chapter covers:

- How to boost your rendering speed by *locking* and *batching*
- How to use *regions* for clipping
- How to manipulate the *colormap*
- How to use the *plane groups*

Related Documentation

This chapter is addressed primarily to programmers who write simple applications using canvas subwindows. For lower level details, see the chapter on *Advanced Imaging* in the *SunView System Programmers Guide*.

The pixwin drawing operations do not directly support high-level graphics operations such as shading, segments, 3-D, etc. If your application requires these, then you should consider some graphics package such as SunGKS, SunCore, or SunCGI. All of these will run in windows (see the *SunCore Reference Manual* and *SunCGI Reference Manual* for more information).

Header Files

The definitions necessary to use pixwins are in the header file `<sunwindow/pixwin.h>`, which is included by `<sunwindow/window_hs.h>`, which in turn is included by default when you include `<suntool/sunview.h>`.

Summary Listing and Tables

To give you a feeling for what you can do with pixwins, the following page contains a list of the available pixwin functions and macros. Many of these are discussed in the rest of this chapter and elsewhere (use the *Index* to check). All are briefly described with their arguments in the pixwin summary tables in Chapter 19, *SunView Interface Summary*:

- the *Pixwin Drawing Functions and Macros* table begins on page 356;
- the *Pixwin Color Manipulation Functions* table begins on page 360.

Pixwin Drawing Functions and Macros

```

pw_batch(pw, n)
pw_batch_off(pw)
pw_batch_on(pw)
pw_batchrop(pw, dx, dy, op, items, n)
pw_char(pw, x, y, op, font, c)
pw_close(pw)
pw_copy(dpw, dx, dy, dw, dh, op, spw, sx, sy)
pw_get(pw, x, y)
pw_get_region_rect(pw, r)
pw_line(pw, x0, y0, x1, y1, brush, tex, op)
pw_lock(pw, r)
pw_pfsysclose()
pw_pfsysopen()
pw_polygon_2(pw, dx, dy, nbds, npts, vlist, op, spr, sx, sy)
pw_polyline(pw, dx, dy, npts, ptlist, mvlist, brush, tex, op)
pw_polypoint(pw, dx, dy, npts, ptlist, op)
pw_put(pw, x, y, value)
pw_read(pr, dx, dy, dw, dh, op, pw, sx, sy)
pw_region(pw, x, y, width, height)
pw_replrop(pw, dx, dy, dw, dh, op, pr, sx, sy)
pw_reset(pw)
pw_rop(pw, dx, dy, dw, dh, op, sp, sx, sy)
pw_set_region_rect(pw, r, use_same_pr)
pw_show(pw)
pw_stencil(dpw, dx, dy, dw, dh, op, stpr, stx, sty, spr, sx, sy)
pw_text(pw, x, y, op, font, s)
pw_traprop(pw, dx, dy, t, op, pr, sx, sy)
pw_ttext(pw, x, y, op, font, s)
pw_unlock(pw)
pw_vector(pw, x0, y0, x1, y1, op, value)
pw_write(pw, dx, dy, dw, dh, op, pr, sx, sy)
pw_writebackground(pw, dx, dy, dw, dh, op)

```

Pixwin Color Manipulation Functions

pw_blackonwhite(pw, min, max)	pw_getcolormap(pw, index, count,
pw_cyclecolormap(pw, cycles, index, count)	red, green, blue)
pw_dbl_access(pw)	pw_getdefaultcms(cms, map)
pw_dbl_flip(pw)	pw_putattributes(pw, planes)
pw_dbl_get(pw, attribute)	pw_putcolormap(pw, index, count,
pw_dbl_release()	red, green, blue)
pw_dbl_set(pw, attributes)	pw_reversevideo(pw, min, max)
pw_getattributes(pw, planes)	pw_setcmsname(pw, cmsname)
pw_getcmsname(pw, cmsname)	pw_whiteonblack(pw, min, max)

7.1. What is a Pixwin?

An image in SunView, whether on the screen or in memory, is composed of dots called *pixels* and is represented internally as a rectangle of such pixels. The *pixrect* structure is the construct used at a low level to access an image and operate on it. You can program at the *pixrect* level to draw on the screen; this is covered in the *Pixrect Reference Manual*.

However, in SunView drawing operations are displayed in a window coexisting on the screen with other, possibly overlapping windows. Except in certain circumstances, drawing operations should be “well-behaved,” meaning that they should not spill over into other windows and they should not be visible in portions of the window which are covered by other windows. The *pixwin* is the interface through which you operate on the pixels in a particular window. It guarantees that the above two conditions will be met.

Each pixel has a value. On a monochrome display the value is 1 or 0, since the pixel can only be on or off, black or white. Such pixels are said to be *1 bit deep*. On a color display each pixel can have several values corresponding to different colors.

7.2. Accessing a Pixwin's Pixels

This section summarizes the functions provided for accessing the pixels of a *pixwin*. Most of the *pw_** functions described in this section are based on corresponding *pr_** routines, which are fully documented in the *Pixrect Reference Manual*. For full discussion of the semantics of a given *pixwin* function, refer to the discussion of the corresponding *pixrect* function in the *Pixrect Reference Manual* and/or the errata/addenda section of the most recent *Release Manual*.

In particular the *pixrect* manual gives useful values for the *op* argument which determines what the result of combining the source and destination pixels will be.

The procedures described in this section will maintain the memory *pixrect* for a retained *pixwin*. That is, they perform their operation on the data in memory, as well as on the screen.

Obtaining the Window's Pixwin

All of these procedures require the *pixwin* of the window you are drawing in as an argument. To draw in a canvas, you use the *pixwin* that is returned by the procedure:

```
Pixwin *pw;
    canvas_pixwin(canvas);
    Canvas canvas;
```

Look at the example in Section 5.1, *Creating and Drawing into a Canvas*, to see how *canvas_pixwin()* is used.

The *pixwin* is also available as the value of the *CANVAS_PIXWIN* attribute of the canvas subwindow.³⁹

³⁹ Aside from the canvas *pixwin*, all windows, regardless of type, have a *pixwin* which is available as the value of *WIN_PIXWIN*. However, most applications should not need to explicitly write pixels into other types of windows.

Write Routines

The following routines allow you to draw areas, backgrounds, vectors, text, polygons, lines, and polylines in a pixwin.

Basic RasterOp Operations

The following are the basic low-level raster operations that draw on the screen. They are common to many imaging systems.

```
pw_write(pw, dx, dy, dw, dh, op, pr, sx, sy)
— or —
pw_rop(pw, dx, dy, dw, dh, op, pr, sx, sy)
    Pixwin *pw;
    int     dx, dy, dw, dh, op, sx, sy;
    Pixrect *pr;
```

`pw_write()` and `pw_rop()` are different names for the same procedure. They perform the indicated rasterop (`op`) from the source `pixrect` to the destination in the `pixwin`. Pixels are written to the rectangle defined by `dx`, `dy`, `dw`, and `dh` in the `pixwin` `pw` using rasterop function `op`. `dx` and `dy` are the position of the top left-hand corner of the rectangle, and `dw` and `dh` are the width and height of the rectangle. They are copied from the rectangle with its origin at `sx`, `sy` in the source `pixrect` pointed to by `pr`.

`pw_write()` is essential for many window system operations such as scrolling a window, drawing frames and borders, and drawing an icon on the screen.

Other Raster Operations

The routines in this section are variations on the basic rasterop routine.

```
pw_writebackground(pw, dx, dy, dw, dh, op)
    Pixwin *pw;
    int     dx, dy, dw, dh, op;
```

`pw_writebackground()` uses a conceptually infinite set of pixels, all of which are set to zero, as the source. It is often used to clear a canvas `pixwin` before drawing a new image.⁴⁰

The following routine draws a pixel of value at (`x`, `y`) in the addressed `pixwin`:

```
pw_put(pw, x, y, value)
    Pixwin *pw;
    int     x, y, value;
```

Using this routine to draw is very slow and should be avoided. If you use it, be sure to read the later sections on *batching* and *locking*.

⁴⁰ Canvases will automatically clear damaged areas if they are set not to be retained, or if the attribute `CANVAS_AUTO_CLEAR` is set. See Chapter 5, *Canvases*, for more information.

There is a similar routine to draw many pixels in a single call.

```
pw_polypoint(pw, dx, dy, npts, ptlist, op)
    Pixwin      *pw;
    int         dx, dy, npts;
    struct pr_pos *ptlist;
    int         op;
```

All `npts` points in the array `ptlist` are drawn in the pixwin `pw` starting at the offset `dx, dy` under the control of the `op` argument.

The next routine draws a vector of pixel value from `(x0, y0)` to `(x1, y1)` in the addressed pixwin using rasterop `op`:

```
pw_vector(pw, x0, y0, x1, y1, op, value)
    Pixwin *pw;
    int     x0, y0, x1, y1, op, value;
```

To replicate a pattern in a `pixrect` onto a `pixwin`, use:

```
pw_replrop(pw, dx, dy, dw, dh, op, pr, sx, sy)
    Pixwin *pw;
    int     dx, dy, dw, dh, op, sx, sy;
    Pixrect *pr;
```

`pw_replrop()` replicates a small “patch” of pattern in a `pixrect` onto an entire `pixwin`. It is often used to draw a patterned background in a window, such as the root gray pattern in `sunview(1)`. Standard patterns, created by `iconedit(1)`, may be found in `/usr/include/images/square_*.pr`.

Text Routines

The following two routines write a string of characters and a single character, respectively, to a `pixwin`, using rasterop `op` as above:

```
pw_text(pw, x, y, op, font, s)
    Pixwin *pw;
    int     x, y, op;
    Pixfont *font;
    char     *s;
```

```
pw_char(pw, x, y, op, font, c)
    Pixwin *pw;
    int     x, y, op;
    Pixfont *font;
    char     c;
```


These text rendering routines are distinguished by their own coordinate system: the destination is given as the left edge and *baseline* of the first character. The left edge does not take into account any *Kerning* (character position adjustment depending on its neighbors), so it is possible for a character to have some pixels to the left of the x-coordinate. The baseline is the y-coordinate of the lowest pixel of characters without descenders, 'L' or 'o' for example, so pixels will frequently occur both above and below the baseline in a string.⁴¹

`font` may be `NULL` in which case the *system font* is used.

The system font is reference counted and shared between software packages. The following routines are provided to open and close the system font:⁴²

```
Pixfont *
pw_pfsysopen()

pw_pfsysclose()
```

The following routine:

```
pw_ttext(pw, x, y, op, font, s)
    Pixwin *pw;
    int      x, y, op;
    Pixfont *font;
    char      *s;
```

is just like `pw_text()` except that it writes *transparent* text. Transparent text writes the shape of the letters without disturbing the background behind it. This is most useful with color pixwins. Monochrome pixwins can use `pw_text()` and a `PIX_SRC | PIX_DST` op, which is faster.

Batching and Stenciling Routines

Applications such as displaying text perform the same operation on a number of pixrects in a fashion that is amenable to global optimization. The batchrop procedure is provided for these situations:

```
pw_batchrop(pw, dx, dy, op, items, n)
    Pixwin *pw;
    int      dx, dy, op, n;
    struct pr_prpos items[];43
```

Stencil operations are like raster ops except that the source pixrect is written through a stencil pixrect which functions as a pixel-by-pixel write enable mask. The indicated raster operation is applied only to destination pixels where the stencil pixrect `stpr` is non-zero; other destination pixels remain unchanged.

⁴¹ A font to be used in `pw_text()` is required to have the same `pc_home.y` and character height for all characters in the font.

⁴² The system font can also be obtained by calling `pf_default()`.

⁴³ The structure of `pr_prpos` is given in Appendix C of the *Pixrect Reference Manual*.

```

pw_stencil(dpw, dx, dy, dw, dh, op, stpr, stx,
           sty, spr, sx, sy)
Pixwin *dpw;
Pixrect *stpr, *spr;
int dx, dy, dw, dh, op, stx, sty, sx, sy;

```

Drawing Polygons

The following macro draws a polygon within a pixwin:

```

pw_polygon_2(pw, dx, dy, nbds, npts, vlist, op, spr, sx, sy)
Pixwin *pw;
int dx, dy, nbds, op, sx, sy;
int npts[];
struct pr_pos *vlist;
Pixrect *spr;

```

You can create a polygon filled with a solid or textured pattern.

Drawing Curved Shapes

`pw_traprop()` is a pixwin operation analogous to `pw_rop()`, which operates on a *trapezon* rather than a rectangle:

```

pw_traprop(pw, dx, dy, t, op, pr, sx, sy)
Pixwin *pw;
struct pr_trap t;
Pixrect *pr;
int dx, dy, op, sx, sy;

```

`pw_traprop()` writes the source pixrect `pr` into the destination pixwin `pw` via the operation `op`. The output is clipped to the trapezon `t`.

Drawing Lines

The following routine draws a solid or textured line between two points with a “brush” of a specified width:

```

pw_line(pw, x0, y0, x1, y1, brush, tex, op)
Pixwin *pw;
int x0, y0, x1, y1, op;
struct pr_brush *brush;
struct pr_texture *tex;

```

There is a similar routine to draw several noncontiguous line segments between a set of points:

```

pw_polyline(pw, dx, dy, npts, ptlist, mvlist, brush, tex, op)
Pixwin *pw;
int dx, dy, npts, op;
struct pr_pos *ptlist;
u_char *mvlist;
struct pr_brush *brush;
struct pr_texture *tex;

```


Read and Copy Routines

The following routines use the pixwin as a source of pixels. To get the value of the pixel at (x, y) in pixwin pw call:

```
int
pw_get(pw, x, y)
    Pixwin *pw;
    int     x, y;
```

To read pixels from a pixwin into a pixrect call:

```
pw_read(pr, dx, dy, dw, dh, op, pw, sx, sy)
    Pixwin *pw;
    int     dx, dy, dw, dh, op, sx, sy;
    Pixrect *pr;
```

This routine reads pixels from pw starting at offset (sx, sy), using rasterop op. The pixels are stored in the rectangle with its origin at dx, dy of width dw and height dh in the pixrect pointed to by pr.

When the destination, as well as the source, is a pixwin, use:

```
pw_copy(dpw, dx, dy, dw, dh, op, spw, sx, sy)
    Pixwin *dpw, *spw;
    int     dx, dy, dw, dh, op, sx, sy;
```

dpw and spw must be the same pixwin. Also, only horizontal or vertical copies are supported.

These read and copy routines fail if they try to read from a portion of a non-retained pixwin which is hidden, and therefore has no pixels. Therefore it is considered advanced usage to call them on a non-retained pixwin; refer to the section entitled *Handling Fixup* in the *SunView 1 System Programmer's Guide*.

7.3. Rendering Speed

Making correct and judicious use of explicit display locking and/or batching is important for getting the best display speed possible.

There are two major impediments to you getting the best possible display rendering speed. The first is *display locking*, which prevents window processes from interfering with each other in several ways:

- Raster hardware may require several operations to complete a change to the display; one process' use of the hardware should be protected from interference by others during this critical interval.
- Changes to the arrangement of windows must be prevented while a process is painting, lest an area be removed from a window as it is being painted.
- A software cursor that the window process does not control (the kernel is usually responsible for the cursor) may have to be removed so that it does not interfere with the window's image.

Display locking is relatively expensive compared to the time it takes to do simple display operations. Thus you can reduce your display time by reducing the number of times that you have to acquire the display lock. The subsection below titled *Locking* explains how to do this.

The second major impediment to maximum display speed is the use of retained pixwins. It is obvious that if you have to write to the screen and to memory for every display operation that it will take longer than writing to only one place. Thus, there is a mechanism, called *pixwin batching* which allows you to write only to memory and then refresh the screen with a quick raster operation from memory. The subsection entitled *Batching* explains how to use batching.

Locking

Locking allows a client program to obtain exclusive use of the display. If the client program does not obtain an explicit lock, the window system will. For example, if your application is going to draw one hundred lines it can either explicitly lock the display once, draw the lines, and unlock explicitly, or it can ignore locking and simply draw the lines. In the latter case, the window system will perform locking and unlocking around each drawing operation, acquiring and releasing the lock one hundred times instead of once.

NOTE *For efficiency's sake, application programs should lock explicitly around a body of screen access operations.*

You can acquire a lock by calling the macro:

```
pw_lock(pw, r)
    Pixwin *pw;
    Rect    *r;
```

`pw` is the pixwin to be used for the output; `r` is the rectangle in the pixwin's coordinate system that bounds the area to be affected. See *The Rect Structure* in Chapter 4, *Using Windows*, for an explanation of the `Rect` structure.

`pw_lock()` blocks if the lock is unavailable (if, for example, another process currently has the display locked).

When the cursor is on the surface where drawing occurs, if the pixwin is locked with `pw_lock()`, sometimes the region in which the cursor rect resides is not drawn to. This results in an empty region (16 x 16 pixels) when the cursor is moved. The image is put to its correct state when it is redisplayed.

Lock operations for a single pixwin may be nested; inner lock operations merely increment a count of locks outstanding and are thus very lightweight. Their affected rectangles must lie within the rectangles affected by the original lock.

To decrement the lock count, call:

```
pw_unlock(pw)
    Pixwin *pw;
```

When the lock count reaches 0, the lock is actually released.

Since locks may be nested, it is possible for a client procedure to find itself, especially in error handling, with a lock which may require an indefinite number of unlocks. To handle this situation cleanly, another routine is provided. The following macro sets `pw`'s lock count to 0 and releases its lock:

```
pw_reset(pw)
    Pixwin *pw;
```


Acquisition of a lock has the following effects:

- If the cursor is in conflict with the affected rectangle, it is removed from the screen. While the screen is locked, the cursor will not be moved in such a way as to disrupt any screen accessing.
- Access to the display is restricted to the process acquiring the lock.
- Modification of the database that describes the positions of all the windows on the screen is prevented.
- The clipping information for the pixwin is validated and, if necessary, updated.
- In the case of a non-retained pixwin with only a single rectangle visible, the internals of the pixwin mechanism can be set up to bypass the pixwin software by going directly to the pixrect level on subsequent display operations.

While it has the screen locked, a process should *not*:

- do any significant computation unrelated to displaying its image.
- invoke any system calls, including other I/O, which might cause it to block.
- invoke any pixwin calls except `pw_unlock()` and those described in the previous section, *Accessing a Pixwin's Pixels*. In any case, the lock should not be held longer than about a quarter of a second, even following all these guidelines.

When a display lock is held for more than two seconds of process virtual time, the lock is broken. However, the offending process is not notified by signal, because a process shouldn't be aborted for this infraction. Instead, a message is displayed on the console.

Batching

Batching allows you to write only to the memory pixrect of a retained pixwin and then refresh the screen with the memory pixrect's contents at specific times. If you do not explicitly batch when using a retained pixwin, the window system will write to both the display and memory on every display operation.

Considering the same example used for locking above, if your application program has a retained pixwin and is going to draw one hundred lines, it can either explicitly start a batch, draw the lines, and end the batch explicitly, or it can ignore batching and simply draw the lines. In the latter case, the window system will draw the lines two hundred times instead of one hundred times.

NOTE *For efficiency's sake, application programs should batch explicitly around a body of screen access operations when using a retained pixwin.*

Two macros are provided to control batching:

```
pw_batch_on(pw)
    Pixwin *pw;

pw_batch_off(pw)
    Pixwin *pw;
```

`pw_batch_on()` starts a batch; `pw_batch_off()` refreshes the screen with the portion of the memory pixrect that has changed. While batching, the pixwin internally maintains a rectangle that describes which pixels in the memory pixrect need to be transferred to the screen at the end of the batch.

NOTE *Don't turn batching on and leave it on, as this causes problems with scrolling. The recommended use is `batch_on()` (draw something in window) `batch_off()`.*

While in the middle of batching, your code might reach a point at which you would like the screen to be updated. The following macro refreshes the screen, but otherwise doesn't change the batching mode:

```
pw_show(pw)
    Pixwin *pw;
```

Unlike locking operations, batch operations for a single pixwin do not nest. Thus, each batching routine in this section affects the batching mode/status.

These three macros — `pw_batch_on()`, `pw_batch_off()` and `pw_show()` — all call the routine `pw_batch()` which actually implements the batching mechanism. You can call `pw_batch()` directly to tell the batching mechanism to refresh the screen after every *n* display operations.

```
pw_batch(pw, kind)
    Pixwin      *pw;
    Pw_batch_type kind;
```

Because the routine does more than one kind of thing, calling it is a little tricky. *kind* is the kind of batching requested. You use the following macro to convert *n*, the number of display operations you want to be batched before a refresh, to a `Pw_batch_type`:

```
#define PW_OP_COUNT(n) ((Pw_batch_type)(n))
```

So, to have batching and ensure the image on-screen is refreshed after every *n* operations, call:

```
pw_batch(pw, PW_OP_COUNT(n));
```

Clients with a group of screen updates to do can gain noticeably by doing the group as a batch. Also, the locking overhead, discussed above, will only be incurred when the screen is refreshed. An example of such a group is displaying a screen full of text, or a series of vectors with pre-computed endpoints.

In considering how to do batching, it's a good idea to be sensitive to how long the user is staring at a blank screen or an old image, and adjust the rate of screen

Locking and Batching Interaction

refresh accordingly.

There are situations in which batching around locking calls makes sense. Consider that

- while batching, locking calls are a no-op;
- if a pixwin is not retained, batching calls are a no-op.

Thus, if your application has a switch to run retained or not, it makes good sense to batch around locking calls. If you batch around locking calls then your application gets the benefit of batching if running retained and the benefit of locking if running non-retained.

Locking around batches, on the other hand, is not very efficient.

7.4. Clipping With Regions

You can use pixwins to clip rectangular regions within a window's own rectangular area. The *region* operation creates a new pixwin that refers to an area within an existing pixwin:

```
Pixwin *
pw_region(pw, x, y, w, h)
    Pixwin *pw;
    int      x, y, w, h;
```

pw is the source pixwin; *x*, *y*, *w* and *h* describe the rectangle to be included in the new pixwin. The upper left pixel in the returned pixwin is at coordinates (0,0); this pixel has coordinates (*x*, *y*) in the source pixwin.

If the source pixwin is retained, the new region will be retained as well. However, the region refers back to the bits of memory *pixrect* of the source pixwin when accessing the image.

To change the size of an existing region, call:

```
int
pw_set_region_rect(pw, r, use_same_pr)
    Pixwin *pw;
    Rect    *r;
    unsigned use_same_pr;
```

The position and size of the region *pw* are set to the rect **r*; a return value of -1 indicates failure. This is more efficient than destroying the old region and creating a new one. The *use_same_pr* flag should be set to 0 if you want a new retained *pixrect* allocated for the region that is the size of the region.

To determine the size of an existing region, call:

```
int
pw_get_region_rect(pw, r)
    Pixwin *pw;
    Rect    *r;
```

*r is set to the size and position of the region pw.

When finished with a region, you should release it by calling: to:

```
pw_close(pw)
    Pixwin *pw;
```

This routine frees any dynamic storage associated with the pixwin, including its retained memory pixrect, if any. If the pixwin has a lock on the screen, it is released.

NOTE You should close any regions before closing the pixwin containing the regions.

7.5. Color

The discussion which follows is divided into three sections:

- *Introduction to Color*, which introduces the concepts of the colormap and colormap segments,
- *Changing the Colormap*, which describes how to change a colormap segment, and
- *Using Color*, which describes how to make color applications compatible with monochrome and grayscale screens, and how to perform smooth animation by using double buffering.

Introduction to Color

Just as there must be arbitration between different windows to decide what is displayed on the screen when several windows overlap, there must likewise be some process of allocation when several windows want to display different sets of many colors all at once. To understand how this works you need to know how color is handled.

The pixels on a color display are not simply on or off; they take many different values for different colors. On all current Sun color displays⁴⁴ each pixel has 8 bits. Such an “8 bit deep” pixel can have any value from 0 to 255. The value in each pixel helps to determine what color appears in that dot on the screen, but it is not in a one-to-one correspondence with the color displayed; otherwise Sun color displays would only be able to display 256 different colors.

The Colormap

Instead, the value of the pixel serves as an index into the *colormap* of the display. The colormap is an array of 256 *colormap entries*. The colormap entry for each index drives the color that is actually displayed for the corresponding pixel value. A colormap entry consists of 8 bits of red intensity, 8 bits of green intensity and 8 bits of blue, packaged into the following structure:

```
struct singlecolor {
    u_char  red, green, blue;
};
```

Hence a Sun color display is capable of displaying over 16 million colors (because each colormap entry has 24 bits) but can only display 256 colors *simultaneously* (because there are only 256 colormap entries).

⁴⁴ See *cgone(4S)*, *cgtwo(4S)* and *cgfour(4S)* in the *UNIX Interface Overview* manual.

A Colormap Example

Suppose that in a group of pixels on the screen, some have the value 0 while others have the value 193. All pixels with the same value will be displayed in the same color. The colormap determines what that color will be. If entry 0 in the colormap of the screen is

```
red = 250; green = 0; blue = 3;
```

then the pixels with a value of 0 will come out bright red. If entry 0 in the colormap is changed to

```
red = 1; green = 8; blue = 2;
```

then the pixels with a value of 0 will immediately change color to an almost-black green. Similarly, entry 193 in the colormap determines what color the pixels with a value of 193 will have.

Changing the Colormap

Because changing the colormap is much faster than redrawing many thousands of pixels with a new value, manipulating the colormap is the basis of many graphics and animation techniques. For examples of programs that manipulate the colormap, run `/usr/demo/suncube` or `/usr/demo/flight`.

Try running **spheresdemo -g** plus another color program at the same time. You will notice that as you move the mouse into the `spheresdemo` window, the colors in the other windows on the display change dramatically. This is because hardware is only capable of displaying 256 colors at once. When two programs that each want to display 256 different colors are run simultaneously, the window system itself must manipulate the colormap. When the cursor enters one of the windows, the window system changes the colormap to use the colors of that window.

Colormap Segments

The window system allows each window to claim a portion of the total available colormap entries, called a *colormap segment*. The colormap segment need not be the same in all windows of a tool: frames and subwindows can have different colormaps, or can share colormaps (see *Sharing Colormap Segments* below). If the total number of entries in all the colormap segments being requested exceeds the limit of 256 at any given time, the window system gives priority to the window under the cursor, and removes segments belonging to other windows as necessary.

The window system loads colormap segments at arbitrary locations within the colormap. To the application program, this indirection is transparent. The routines that access a `pixwin`'s pixels do not distinguish between windows which use colormap segments and those which use the entire colormap.

NOTE *While you can have multiple `pixwins` within a window, there is only one colormap segment per window. A separate colormap for each `pixwin` in a window is not supported. This limitation should only be of interest if you are using `pixwin` regions (described in the *SunView System Programmer's Guide*).*

- Background and Foreground** Every colormap segment has two distinguished values, its *background* and *foreground*. The background color is defined as the value at the first position of a colormap segment; the foreground color is the value at the last position.
- Default Colormap Segment** The first pixwin created for a window sets the background and foreground of the window to be those of the *default colormap segment*. This is the monochrome colormap segment defined in
- `<sunwindow/cms_mono.h>`. Subsequent pixwins created for the window inherit the background and foreground of the window.
- Changing Colors from the Command Line** The user can modify the default colormap for all applications by invoking `-sunview` with the `-F` and `-B` command line arguments.⁴⁵ The user can also change the default colormap segment on a per-application basis by invoking the application with certain flags. The `-Wf` flag sets the foreground color, `-Wb` sets the background color, and `-Wg` specifies that the colormap of the frame will be inherited by the frame's subwindows.
- The equivalent frame attributes for these flags are `FRAME_FOREGROUND_COLOR`, `FRAME_BACKGROUND_COLOR`, and `FRAME_INHERIT_COLORS`.
- Sharing Colormap Segments** It is possible for different processes to share a single colormap segment. For some applications, you want to guarantee that your colormap segment is not shared by another process. For example, a colormap segment to be used for animation, as described later in the section on *Double Buffering*, should not be shared. The way to ensure that a colormap segment will not be shared by another window is to give it a unique name. A common way to generate a unique name is to append the process' id to a more meaningful string that describes the usage of the colormap segment.
- If a colormap segment's usage is static in nature, then it pays to use a shared colormap segment definition, since colormap entries are scarce. Windows, in the same or different processes, can share the same colormap by referring to it by the same name.
- There are three basic types of shared colormap segments:
- A colormap segment used by a single program. Sharing occurs when multiple instances of the same program are running. An example of such a program is a color terminal emulator in which the terminal has a fixed selection of colors.
 - A colormap segment used by a group of highly interrelated programs. Sharing occurs whenever two or more programs of this group are running at the same time. An example of such a group is a series of CAD/CAM programs in which it is common to have multiple programs running at the same time.

⁴⁵ This is not true for a Sun-3/110 and other machines with `cgfour` frame buffers, due to their use of an overlay plane to implement most monochrome windows.

- A colormap segment used by a group of unrelated programs. Sharing occurs whenever two or more programs of this group are running. An example of such a colormap segment is the default colormap, CMS_MONOCHROME, defined in <sunwindow/cms_mono.h>. Other common useful colormap segment definitions that you can use and share with other windows include cms_rgb.h, cms_grays.h, cms_mono.h, and cms_rainbow.h, found in <sunwindow/cms_*.h>.

Example: *showcolor*

The program on the following page shows the actual colors in the display's colormap. It should help you see how the window system manages the colormap. Run this program soon after bringing up sunview, then run several color graphics programs such as the demos mentioned earlier. Try bringing up different windows with different foreground and background colors, as in:

```
% shelltool -Wf 23 182 48 -Wb 255 200 230 -Wg
```

```
/*
 *
 *                               showcolor.c
 *   Draw a grey ramp that graphically shows the colormap
 *   segment activity of the environment when the cursor
 *   is NOT in the canvas of this tool.
 */

#include <suntool/sunview.h>
#include <suntool/canvas.h>

#define CMS_SIZE      256
#define CAN_HEIGHT    10

main(argc, argv)
    char    **argv;
{
    Frame      frame;
    Canvas     canvas;
    register Pixwin *pw;
    register int i;
    u_char     red[CMS_SIZE],
               green[CMS_SIZE],
               blue[CMS_SIZE];

    /* Create frame and canvas */
    frame = window_create(0, FRAME,
                          FRAME_LABEL, argv[0],
                          FRAME_ARGS, argc, argv,
                          0);
    canvas = window_create(frame, CANVAS,
                           WIN_HEIGHT, CAN_HEIGHT,
                           WIN_WIDTH, 2 * CMS_SIZE,
                           0);

    window_fit(frame);
    pw = canvas_pixwin(canvas);

    /* Initialize colormap to grey ramp */
```

```

for (i = 0; i < CMS_SIZE; i++)
    red[i] = green[i] = blue[i] = i;
pw_setcmsname(pw, "showcolor");
pw_putcolormap(pw, 0, CMS_SIZE, red, green, blue);

/* Draw ramp of colors */
for (i = 0; i < CMS_SIZE; i++)
    pw_rop(pw, i*2, 0, 2, CAN_HEIGHT,
           PIX_SRC | PIX_COLOR(i), (Pixrect *)0, 0, 0);
window_main_loop(frame);
exit(0);
}

```

Manipulating the Colormap

The following sections document the routines that implement the techniques described above.

To change a window's colormap segment, you must:

1. Name the colormap segment with `pw_setcmsname()`.
2. Set the size of the segment by loading the colors with `pw_putcolormap()`.

It is important that these two steps happen in order and together. The call to `pw_setcmsname()` does not take effect until you write at least one color value into the colormap with `pw_putcolormap()`.

You set and retrieve the name of a colormap segment with these two functions:

```

pw_setcmsname(pw, name)
    Pixwin *pw;
    char    name[CMS_NAMESIZE];

pw_getcmsname(pw, name)
    Pixwin *pw;
    char    name[CMS_NAMESIZE];

```

If you set the foreground and background colors (which are entries `count - 1` and `0` in the colormap segment, respectively) to the same color, the system will change them to the foreground and background colors of `sunview`. In other words, you are prevented from making the foreground and background colors of a `pixwin` indistinguishable.

Setting the name resets the colormap segment to a NULL entry. After calling `pw_setcmsname()`, you must immediately call `pw_putcolormap()` to set the size of the colormap segment and load it with the actual colors desired. `pw_putcolormap()` and the corresponding routine to retrieve the colormap's state, `pw_getcolormap()`, are defined as follows:


```

pw_putcolormap(pw, index, count, red, green, blue)
    Pixwin      *pw;
    int          index, count;
    unsigned char red[], green[], blue[];

```

```

pw_getcolormap(pw, index, count, red, green, blue)
    Pixwin      *pw;
    int          index, count;
    unsigned char red[ ], green[ ], blue[ ];

```

`pw_putcolormap` loads the count elements of the pixwin's colormap segment starting at index (zero origin) with the first count values in the three arrays.

The first time `pw_putcolormap()` is called after calling `pw_setcmsname()`, the count parameter defines the size of the colormap segment. The size of a colormap segment must be a power of 2, and can't be changed unless `pw_setcmsname()` is called with another name. You can call `pw_putcolormap()` subsequently to modify a subrange of the colormap — use a larger value for index and a smaller value for count.

NOTE *If you attempt to install a colormap segment that is not a power of 2, your colormap segment has a high likelihood of taking up too much space. This means that the screen will flash when you move the cursor into the window with this odd sized colormap.*

In Appendix A, *Example Programs*, there is a program called *coloredit* which uses `pw_putcolormap()` to change the colors of its subwindows as the user adjusts sliders for red, green and blue.

Cycling the Colormap

A utility is provided to make it easy to cycle colormap entries:

```

pw_cyclecolormap(pw, cycles, index, count)
    Pixwin *pw;
    int     cycles, index, count;

```

Starting at index, the count entries of the colormap associated with the pixwin's window are rotated among themselves for cycles. A cycle is defined as number of shifts it takes one entry to move through every position once.

To see an example of colormap cycling, run `jumpdemo (6)` with the `-c` option.

If you are going to cycle one of the common colormap segment definitions, you should give the colormap a unique name, otherwise the colormap of other applications will change as well.

Miscellaneous Utilities

The following utilities are provided as convenient ways to set the foreground and background colors to common settings. `min` should be the first entry in the colormap segment, representing the background color. `max` should be the last entry, representing the foreground color.

```

pw_reversevideo(pw, min, max)
    Pixwin      *pw;
    int         min, max;

pw_blackonwhite(pw, min, max)
    Pixwin      *pw;
    int         min, max;

pw_whiteonblack(pw, min, max)
    Pixwin      *pw;
    int         min, max;

```

On a monochrome display, these calls don't take effect until you write to the pixwin. On a color display, they take effect immediately.

Using Color

This section gives some notes on the use of color by cursors and menus, how to make color applications compatible with monochrome and grayscale screens, and how to use double buffering for smooth animation.

Cursors and Menus

Cursors appear in the foreground color, the last color in the pixwin's colormap.

Menus and prompts use *fullscreen access*, covered in Chapter 12, *Menus and Prompts*, of the *SunView 1 System Programmer's Guide*. Fullscreen access saves the colors in the first and last entries of the *screen's* colormap, puts in the foreground and background colors, and displays the menu or prompt. This means that depending on where your application's colormap segment resides in the screen's colormap, some colors in your tool may change whenever menus or prompts are put up. You can allow for this by making the background and foreground colors in your colormap segment the same as the screen's background and foreground.

There are other menu/cursor "glitches" that occur when running applications on frame buffers which support multiple plane groups. These are covered in the later section on *Multiple Plane Groups*.

Is My Application Running on a Color Display?

None of the colormap manipulations described in this chapter causes an error if run on a monochrome display. All colors other than zero map to the foreground color, so if your application displays colored objects on a background of zero, they will appear as black objects on a white foreground on a monochrome display⁴⁶. The window system detects and prevents the foreground and background being the same color on color displays.

However, you may want to determine at run time whether your application has a color or monochrome display available to it. For example, when displaying a chart, you may want to use patterns if colors are not available. You can determine whether the display is color or monochrome by finding out how deep the pixels are. Each pixwin includes a pointer to a pixrect which represents its pixels on the screen. Pixrects, in turn, have a depth field which holds the number of bits

⁴⁶ Unless you are running with black and white inverted, using the `-i` option to `sunview`.

per screen pixel. Thus

```
Pixwin *pw;

int depth = pw->pw_pixrect->pr_depth;
```

will have a value of 1 for windows displayed on monochrome devices, and a value greater than 1 for color screens. Currently, all Sun color displays have 8 bits per pixel.

Simulating Grayscale on a Color Display

There is no way to tell if your application is running on a grayscale monitor, since it runs off the same color board. The grayscale monitor is usually driven from the red output of the color board, so if two colors have different green and blue values but the same red value, they will show up the same on a grayscale display.

To see how your color application will look on a grayscale monitor, temporarily set your colormap segment so that the green and blue components of each colormap entry are the same as the red component. This will simulate the grayscale display on a color monitor.

Software Double Buffering

Sometimes you want to rapidly display different images in an application. If you just use the pixwin write operations to display the new image, the redrawing of the pixels will be perceptible to the user, even though the operations are fast. Instead, you can use a technique called *software double-buffering*.

As we have seen, on a color display, there are 8 bits associated with each pixel. If you are not using 256 shades at once, then some of these bits are unused. What you would like to do is to store values for two or more different images in these 8 bits, but only display one set of values at a time.

The first goal can easily be accomplished using the `pw_putattributes()` routine to restrict writes to a particular set of planes:

```
pw_putattributes(pw, planes)
    Pixwin *pw;
    int     *planes;
```

`planes` is a bitplane access enable mask. Only those bits of the pixel corresponding to a 1 in the same bit position of `*planes` will be affected by pixwin operations. If `planes` is NULL, that attribute value will not be written.

A corresponding routine is provided to retrieve the value of the access enable mask:

```
pw_getattributes(pw, planes)
    Pixwin *pw;
    int     *planes;
```

NOTE Use `pw_putattributes()` with care, as it changes the internal state of the `pixwin`. The correct usage is to first save the existing bitplane mask by calling

`pw_getattributes()`, then call `pw_putattributes()`, then, when done, restore the initial state by calling `pw_putattributes()` with the saved mask.

The second goal — only displaying what is in some of the planes — is trickier. There is no way to tell the hardware to only look at the values in some of the planes to determine the colors to show.

What you do instead is modify the colormap so that only values in certain planes of the colormap change the color on the display, so in effect only those planes are visible. For example, to display two different four-color images you could use the colormap shown in the following table.

Table 7-1 *Sample Colormap to Isolate Planes*

Pixel Value	Colormap A (Only upper planes are "visible")	Colormap B (Only lower planes are "visible")
0 0 0 0	blue	blue
0 0 0 1	blue	red
0 0 1 0	blue	green
0 0 1 1	blue	pink
0 1 0 0	red	blue
0 1 0 1	red	red
0 1 1 0	red	green
0 1 1 1	red	pink
1 0 0 0	green	blue
1 0 0 1	green	red
1 0 1 0	green	green
1 0 1 1	green	pink
1 1 0 0	pink	blue
1 1 0 1	pink	red
1 1 1 0	pink	green
1 1 1 1	pink	pink

From the above table, you can see that if colormap A is set (using `pw_putcolormap()`), then no matter what the value in the two lower planes, the color displayed is the same; the value in the upper two planes alone controls the color. So, if you use this colormap while only enabling the two lower planes (by passing `pw_putattributes()` the value 3), then the values you write into the lower planes won't change what is shown.

When you switch to colormap B, the situation is reversed. Only the values in the lower planes affect what is visible. You would then pass `pw_putattributes()` the value 12 to write to the upper two planes. The two sets of colors need not be the same, so you can switch between two different-colored images.

Using Software Double Buffering For Smooth Animation

You would use the same technique to switch between more images and/or to display more colors. You can display two different images, each with 16 different colors, or 8 different monochrome images, or values in between.

One application of the above technique is to provide smooth animation. To move an image across the screen, you must draw it in one location, erase it, and redraw it in another. Even on a fast system, the erasing and redrawing is visible. You'd like the object to immediately appear in its new position, without disappearing momentarily. You can do this by alternating two colormaps so that the object disappears in its old location and reappears in a new one. This is called *software double-buffering*, because you are using the display planes as alternating buffers; as you write to one set of planes, the other set of planes is displayed.

The colormaps in the table on the preceding page come from the software program *animatecolor* in Appendix A, *Example Programs*. This program uses software double buffering to animate some squares. The routines it uses to create the two colormaps and swap between them are complicated, but can be reused in more sophisticated graphics applications.

Hardware Double-Buffering

The following routines will allow programs to do true hardware double-buffering on the Thecg5board. on the device driver interface, refer the the `cgtwo(4S)` manual page.⁴⁷ color framebuffer and on future framebuffers that support double-buffering.

Double-buffering is treated as an even scarcer resource than colormaps, since only one window can be truly double-buffered at any one time. The cursor controls which window will flip the display buffers. Applications are able to run the same code on non-double-buffered displays and it will be as if the double-buffering calls were never made. The following code fragment contains prototypical application code.

```
Rect    rectangle;
Pixwin  *pw;
rectangle.r_left= ...;
...
if (!pw_dbl_get (pw, PW_DBL_AVAIL))
{ ... if program cares ... }
pw_dbl_access (pw);
while (rendering_frames) {
    ... calculate one frame ...
    pw_lock (pw, &rectangle);
    ... render one frame ...
    ... may include unlocks and locks ...
    pw_dbl_flip (pw);
    pw_unlock (pw);
}
pw_dbl_release (pw);
```

⁴⁷ The cg5 board is binary compatible with both the Sun-3 Color Board and the Sun-2 Color Board. cg5 is necessary for hardware double-buffering.

The notion of the “active” double-buffering window is important. There is at most one active window at a time. If the cursor is in a double-buffering window, then the window is the active double-buffering window. If the cursor leaves the active window, that window remains active until the cursor enters another double-buffering window. If the active double-buffering window dies, goes iconic, or becomes totally obscured, and the cursor is not left in a double-buffering window, then the top-most visible double-buffering window becomes the active window (if there is one).

Only the active window will be allowed to write to a single buffer. All other windows write to both buffers, so that when the display flips to the other buffer, their contents remain unchanged. The notion of active will change only during a `pw_dbl_flip()` call.

`pw_dbl_access()` which resets the window’s data structure so that first frame will be rendered to the background. The very first double buffer sets both READ and WRITE to the background. `pw_dbl_access()` should only be called when ready to actively animate:

```
pw_dbl_access (pw)
    Pixwin *pw;
```

If the pixwin’s window has not been accessed for double-buffering then there is no change, and both buffers will be written to.

If the window is marked as accessible for double-buffering and the window is “active”, then the frame double-buffering control to whatever this window requested with its last `pw_dbl_set()` call. If there was no `pw_dbl_set()` call, then set WRITE and READ to the background. Change the frame buffer double-buffering control bits to reflect this.

If the window is accessible for double-buffering then potentially flip the display. The display is flipped only if the window is “active”: `pw_dbl_flip()` determines if its window has become active:

```
pw_dbl_flip(pw)
    Pixwin *pw;
```

The flip can be done inside or outside of a lock region although it may be preferable to place inside a lock region just before an unlock so that calculations for the next frame can proceed even if another window momentarily grabs the lock. The flip from one buffer to another is synchronized with the display’s vertical retrace.

The procedure

```
pw_dbl_release(pw)
    Pixwin *pw;
```

signifies the end of double-buffering by the window associated with the pixwin. Call `pw_dbl_release()` as soon as your program has completed a section of active animation. This procedure will copy the foreground buffer to the background. Because of this, it is important to leave the animation loop after a `pw_dbl_flip()` has been done and before drawing the next frame has started. Otherwise, the window will contain an incomplete buffer image after the release.

SunView provides the ability for an actively double-buffering window to write to both buffers. For example, the instrument gauge readings can be set in a real-time simulator. If `pw` is not the active double buffer, the frame buffer control bits are not changed. The procedure and the attributes that it may use are discussed below.

```
pw_dbl_set(pw, attributes)
    Pixwin      *pw;
    <attribute-list> attributes;
```

Table 7-2 *Pixwin-Level set Attributes*

Attribute	Possible Values to set
PW_DBL_WRITE	PW_DBL_FORE, PW_DBL_BACK, PW_DBL_BOTH
PW_DBL_READ	PW_DBL_FORE, PW_DBL_BACK

The attribute value returned from `pw_dbl_get()` does not reflect the true state of double buffering hardware. This is especially true if the active double buffer is not this `pixwin`. The procedure and the attributes that it uses are given below.

```
pw_dbl_get(pw, attribute)
    Pixwin      *pw;
    Pw_dbl_attribute attribute;
```

Table 7-3 *Pixwin-Level get Attributes*

Attribute	Possible Values Returned
PW_DBL_AVAIL	PW_DBL_EXISTS
PW_DBL_DISPLAY	
PW_DBL_WRITE	PW_DBL_FORE, PW_DBL_BACK, PW_DBL_BOTH
PW_DBL_READ	PW_DBL_FORE, PW_DBL_BACK

7.6. Plane Groups and the `cgfour` Frame Buffer

The Sun-3/110, Sun-3/60, and Sun-4/110 color machines use the `cgfour(4s)`⁴⁸ frame buffer, which supports multiple "plane groups." Each displays either 24-bit color or black and white. In the former case its color is determined by a value in an 8-bit color buffer; in the latter case, a monochrome buffer called the *overlay plane*.

Whether the pixel displays in color or black/white is controlled by the value for the pixel in the *enable plane*, a third plane. If the value in the enable plane is not set, then the 8-bit deep value in the color buffer is passed to the circuitry that produces the color from the lookup table. If it is set, then the overlay plane determines the pixel's color (black or white). The effect is like having a color and monochrome display in one, with the enable plane determining which is shown in each pixel.

⁴⁸ Read the `cgfour(4s)` manual page for more information on this frame buffer architecture.

In fact, in the color Sun-3/60 and Sun-4/110 plane group implementations, you can set the colors in the overlay plane to other than black and white. There are only two colors in the overlay plane since it is only one-bit deep, but they can have colors other than black and white assigned to them.

Such sets of buffers are referred to as *plane groups*.

SunView and Plane Groups

At the pixrect level it is possible to manipulate the three plane groups of multiple plane group framebuffers directly. At the SunView level, some decisions have been made for you. Raster operations in the overlay plane are faster than in the color plane, so SunView objects which only use the foreground and background colors such as frames, text subwindows, panels, cursors, menus, etc. all try to run in the overlay plane. If you set the foreground and background explicitly using the techniques explained in *Changing Colors from the Command Line* above, or if you have told `sunview` to run in the color buffer only by giving it the command line argument `-8bit_color_only`, then these objects will run in the color plane.

However, canvases and graphics subwindows default to using the color plane group whenever possible, on the assumption that you want to draw in color. If this is not the case, then you may find that your application runs faster if you hint to these subwindows to use the overlay plane:

- For canvases, set the attribute `CANVAS_FAST_MONO`, either when creating the canvas or later, as in:

```
window_set(canvas, CANVAS_FAST_MONO, TRUE, 0);
```

If your application uses scrollbars, then you need to set `CANVAS_FAST_MONO` before you create the canvas' scrollbars, since they share the canvas' pixwin.

- For graphics subwindows in old-style SunWindows applications, use the pixwin call `pw_use_fast_monochrome(pw)` as follows:

```
pw_use_fast_monochrome(gfx->gfx_pixwin);
```

Both calls affect only multiple plane group displays, so it is safe and desirable to put them in any Sun application that uses monochrome canvases or graphics subwindows. Again, if the user gives the appropriate command line arguments, canvases and graphics subwindows will run in the color plane regardless of these calls.

"Glitches" Visible when Using Plane Groups

For performance reasons, the cursor image is only written in the plane group of the window under it. So, if the cursor's *hot spot* is in a black and white window in the overlay plane and there is an adjacent color window, that part of its image that would lie over the color window is invisible, since it is drawn in the overlay plane but the enable plane is still showing the value in the color buffer. The same disappearance applies in the reverse situation.

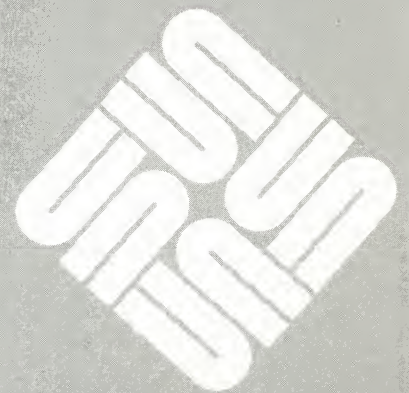
When menus are drawn, the enable plane is set so that they are visible.

sunview and Plane Groups

It is possible to direct `sunview(1)` to only use the color buffer or the overlay plane; it is also possible to start up a second copy of `sunview` in the other plane group, and switch between them using `switcher(1)` or `adjacentscreens(1)`. Consult these programs' manual pages for more information.

Text Subwindows

Text Subwindows	129
Summary Tables	129
8.1. Text Subwindow Concepts	132
Creating a Subwindow	132
Attribute Order	132
Determining a Character's Position	132
Getting a Text Selection	132
Editing a Text Subwindow	132
8.2. Loading a File	133
Checking the Status of the Text Subwindow	133
Textsw_status Value	133
8.3. Writing to a Text Subwindow	134
Insertion Point	135
Positioning to End of Text	135
8.4. Reading from a Text Subwindow	135
8.5. Editing the Contents of a Text Subwindow	136
Removing Characters	136
Emulating an Editing Character	136
Replacing Characters	137
The Editing Log	138
Which File is Being Edited?	138
Interactions with the File System	138
8.6. Saving Edits in a Subwindow	139



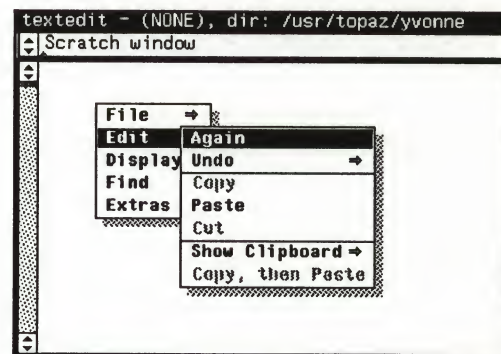
Storing Edits	139
Discarding Edits	139
8.7. Setting the Contents of a Text Subwindow	140
TEXTSW_FILE_CONTENTS	140
TEXTSW_CONTENTS	140
TEXTSW_INSERT_FROM_FILE	141
8.8. Positioning the Text Displayed in a Text Subwindow	141
Screen Lines and File Lines	141
Absolute Positioning	142
Relative Positioning	142
How Many Screen Lines are in the Subwindow?	143
Which File Lines are Visible?	143
Guaranteeing What is Visible	143
Ensuring that the Insertion Point is Visible	143
8.9. Finding and Matching a Pattern	144
Matching a Span of Characters	144
Matching a Specific Pattern	144
8.10. Marking Positions	145
8.11. Setting the Primary Selection	147
8.12. Dealing with Multiple Views	147
8.13. Notifications from a Text Subwindow	148

Text Subwindows

This chapter describes the text subwindow package, which you can use by including the file `<suntool/textsw.h>`.

Figure 8-1 is a text subwindow. A text subwindow allows a user or client to display and edit a sequence of ASCII characters. These characters are stored in a file or in primary memory. Its features range from inserting into a file to searching for and replacing a string of text or a character.

Figure 8-1 *Text Subwindow*



Summary Tables

To give you a feeling for what you can do with text subwindows, overleaf there is a list of the available text subwindow attributes and functions. Many of these are discussed in the rest of this chapter and elsewhere (use the *Index* to check). All are briefly described with their arguments in the text subwindow summary tables in Chapter 19, *SunView Interface Summary*:

- the *Text Subwindow Attributes* table begins on page 366;
- the *Textsw_action Attributes* table begins on page 370;
- the *Textsw_status Values* table begins on page 371;
- the *Text Subwindow Functions* table begins on page 372.

Text Subwindow Attributes

TEXTSW_ADJUST_IS_PENDING_DELETE	TEXTSW_INSERT_FROM_FILE
TEXTSW_AGAIN_RECORDING	TEXTSW_INSERT_MAKES_VISIBLE
TEXTSW_AUTO_INDENT	TEXTSW_INSERTION_POINT
TEXTSW_AUTO_SCROLL_BY	TEXTSW_LEFT_MARGIN
TEXTSW_BLINK_CARET	TEXTSW_LENGTH
TEXTSW_BROWSING	TEXTSW_LINE_BREAK_ACTION
TEXTSW_CHECKPOINT_FREQUENCY	TEXTSW_LOWER_CONTEXT
TEXTSW_CLIENT_DATA	TEXTSW_MEMORY_MAXIMUM
TEXTSW_CONFIRM_OVERWRITE	TEXTSW_MENU
TEXTSW_CONTENTS	TEXTSW_MODIFIED
TEXTSW_CONTROL_CHARS_USE_FONT	TEXTSW_MULTI_CLICK_SPACE
TEXTSW_DISABLE_CD	TEXTSW_MULTI_CLICK_TIMEOUT
TEXTSW_DISABLE_LOAD	TEXTSW_NOTIFY_PROC
TEXTSW_EDIT_COUNT	TEXTSW_READ_ONLY
TEXTSW_FILE	TEXTSW_SCROLLBAR
TEXTSW_FILE_CONTENTS	TEXTSW_STATUS
TEXTSW_FIRST	TEXTSW_STORE_CHANGES_FILE
TEXTSW_FIRST_LINE	TEXTSW_STORE_SELF_IS_SAVE
TEXTSW_HISTORY_LIMIT	TEXTSW_UPDATE_SCROLLBAR
TEXTSW_IGNORE_LIMIT	TEXTSW_UPPER_CONTEXT

Textsw_action Attributes

TEXTSW_ACTION_CAPS_LOCK	TEXTSW_ACTION_TOOL_CLOSE
TEXTSW_ACTION_CHANGED_DIRECTORY	TEXTSW_ACTION_TOOL_DESTROY
TEXTSW_ACTION_EDITED_FILE	TEXTSW_ACTION_TOOL_QUIT
TEXTSW_ACTION_EDITED_MEMORY	TEXTSW_ACTION_TOOL_MGR
TEXTSW_ACTION_FILE_IS_READONLY	TEXTSW_ACTION_USING_MEMORY
TEXTSW_ACTION_LOADED_FILE	

Text Subwindow Functions

```
textsw_add_mark(textsw, position, flags)
textsw_append_file_name(textsw, name)
textsw_delete(textsw, first, last_plus_one)
textsw_edit(textsw, unit, count, direction)
textsw_erase(textsw, first, last_plus_one)
textsw_file_lines_visible(textsw, top, bottom)
textsw_find_bytes(textsw, first, last_plus_one, buf, buf_len, flags)
textsw_find_mark(textsw, mark)
textsw_first(textsw)
textsw_index_for_file_line(textsw, line)
textsw_insert(textsw, buf, buf_len)
textsw_match_bytes(textsw, first, last_plus_one,
                   start_sym, start_sym_len, end_sym, end_sym_len, field_flag)
textsw_next(textsw)
textsw_normalize_view(textsw, position)
textsw_possibly_normalize(textsw, position)
textsw_remove_mark(textsw, mark)
textsw_replace_bytes(textsw, first, last_plus_one, buf, buf_len)
textsw_reset(textsw, x, y)
textsw_save(textsw, x, y)
textsw_screen_line_count(textsw)
textsw_scroll_lines(textsw, count)
textsw_set_selection(textsw, first, last_plus_one, type)
textsw_store_file(textsw, filename, x, y)
```


8.1. Text Subwindow Concepts

Creating a Subwindow

This section introduces the basic concepts of a text subwindow.

You create a text subwindow the same way that you create any SunView window object, by calling the window creation routine with the appropriate type parameter:

```
Textsw textsw;
textsw = window_create(base_frame, TEXTSW, attributes, 0);
```

The *attributes* in the above call constitute an attribute list which is discussed in a Section later in this chapter, titled *Attribute-based Functions*.

Attribute Order

Most attributes are orthogonal; thus you usually need not worry about their order. However, in a few cases the attributes in a list may interact, so you need to be careful to specify them in a particular order. Such cases are noted in the sections which follow.⁴⁹ In particular, you must pass TEXTSW_STATUS first in any call to window_create() or window_set() if you want to find the status after setting some other attribute in the same call.

Determining a Character's Position

The contents of a text subwindow are a sequence of characters. At any moment, each character can be uniquely identified by its position in the sequence (type Textsw_index). Editing operations, such as inserting and deleting text, cause the index of any particular character to change over time. The valid indices are 0 through length-1 inclusive, where length is the number of characters currently in the text subwindow, returned by the TEXTSW_LENGTH attribute.

The text subwindow has a notion of the current index after which the next character will be inserted at any given moment. This is called the *insertion point*. A caret is drawn on the screen immediately after this index to give the user a visual indication of the insertion point.

Getting a Text Selection

A text selection is made by the user, and it is indicated on the screen with reverse-video highlighting. A text subwindow function or procedure is not used to determine which window has the current selection or to retrieve information contained in a text subwindow. Instead, these functions are carried out by the Selection Service. For an example of how this is done, refer to Section 16, *The Selection Service*.

Editing a Text Subwindow

A text subwindow may be edited by the user, or by a client program. When you create a text subwindow, by default the user can edit it. By using the special attributes discussed in this section, the client program can edit the subwindow. These edits are then stored in /tmp/textProcess-id.Counter.

The following five sections explain the functions and attributes that you will use to load, read, write, edit, and finally save a text file.

⁴⁹ For a discussion of attribute ordering in general, see Section 4.8, *Attribute Ordering*.

8.2. Loading a File

You can load a file into a textsw by using `TEXTSW_FILE`, as in:

```
window_set(textsw, TEXTSW_FILE, file_name, 0);
```

Keep in mind, that if the existing text has been edited, then these edits will be lost. To avoid such loss, first check whether there are any outstanding edits by calling:

```
window_get(textsw, TEXTSW_MODIFIED)
```

The above call to `window_set()` will load the new file with the text positioned so that the first character displayed has the same index as the first character that was displayed in the previous file — which is probably not what you want. To load the file with the first displayed character having its index specified by position, use the following:

```
window_set(textsw, TEXTSW_FILE, file_name,
           TEXTSW_FIRST, position, 0);
```

NOTE *The order of these attributes is important. Because attributes are evaluated in the order given, reversing the order would first reposition the existing file, then load the new file. This would cause an unnecessary repaint, and mis-position the old file, if it was shorter than position. For a full discussion of attribute ordering, see Section 8.5.*

Checking the Status of the Text Subwindow

Both of the above calls blindly trust that the load of the new file was successful. This is, in general, a bad idea. To find out whether the load succeeded, and if not, why not, use the following call:

```
window_set(textsw,
           TEXTSW_STATUS, &status,
           TEXTSW_FILE, file_name,
           TEXTSW_FIRST, position,
           0);
```

where `status` is declared to be of type `Textsw_status`.

NOTE *The `TEXTSW_STATUS` attribute and handle must appear in the attribute list before the operation whose status you want to determine.*

Textsw_status Value

The valid values for such a variable are enumerated in the following table, where the common prefix `TEXTSW_STATUS_` has been removed. For example, `OKAY` in the table is actually `TEXTSW_STATUS_OKAY`.

Table 8-1 Textsw_status Values

<i>Value</i>	<i>Description</i>
TEXTSW_STATUS_OKAY	The operation encountered no problems.
TEXTSW_STATUS_BAD_ATTR	The attribute list contained an illegal or unrecognized attribute.
TEXTSW_STATUS_BAD_ATTR_VALUE	The attribute list contained an illegal value for an attribute, usually an out of range value for an enumeration.
TEXTSW_STATUS_CANNOT_ALLOCATE	A call to <code>calloc(2)</code> or <code>malloc(2)</code> failed.
TEXTSW_STATUS_CANNOT_OPEN_INPUT	The specified input file does not exist or cannot be accessed.
TEXTSW_STATUS_CANNOT_INSERT_FROM_FILE	The operation encountered a problem when trying to insert from file.
TEXTSW_STATUS_OUT_OF_MEMORY	The operation ran out of memory while editing in memory.
TEXTSW_STATUS_OTHER_ERROR	The operation encountered a problem not covered by any of the other error indications.

8.3. Writing to a Text Subwindow

To insert text into a text subwindow at the current insertion point, call:

```
Textsw_index
textsw_insert(textsw, buf, buf_len)
    Textsw  textsw;
    char    *buf;
    int     buf_len;
```

The return value is the number of characters actually inserted into the text subwindow. This number will equal `buf_len` unless either the text subwindow has had a memory allocation failure, or the portion of text containing the insertion point is read only. The insertion point is moved forward by the number of characters inserted.

NOTE *This routine does not do terminal-style interpretation of the input characters.* Thus "editing" characters (such as CTRL-H or DEL for character erase, etc.) are simply inserted into the text subwindow rather than performing edits to the existing contents of the text subwindow. In order to do terminal-style emulation, you must pre-scan the characters to be inserted, and invoke `textsw_edit()` where appropriate, as described in the next section.

Insertion Point

The attribute `TEXTSW_INSERTION_POINT` is used to interrogate and to set the insertion point. For instance, the following call determines where the insertion point is:

```
Textsw_index point;
point = (Textsw_index)window_get(textsw,
                                TEXTSW_INSERTION_POINT);
```

whereas the following call sets the insertion point to be just before the third character of the text:

```
window_set(textsw, TEXTSW_INSERTION_POINT, 2, 0);
```

Positioning to End of Text

To set the insertion point at the end of the text, set `TEXTSW_INSERTION_POINT` to the special index `TEXTSW_INFINITY`.

NOTE *This call does not ensure that the new insertion point will be visible in the text subwindow, even if the `TEXTSW_INSERT_MAKES_VISIBLE` attribute is `TRUE`. To guarantee that the caret will be visible afterwards, you should call `textsw_possibly_normalize()`.*

8.4. Reading from a Text Subwindow

Many applications that incorporate text subwindows never need to read the contents of the text directly from the text subwindow. Often, this is because the text subwindow is only being used to display text to the user.

Even when the user is allowed to edit the text, some applications simply wait for the user to perform some action that indicates that all of the edits have been made. They then use either `textsw_save()` or `textsw_store_file()` to place the text in the file. The text can then be read via the usual file input utilities, or the file itself can be passed off to another program.

It is, however, useful to be able to directly examine the text in the text subwindow. You can do this using the `TEXTSW_CONTENTS` attribute. The following code fragment illustrates how to use `TEXTSW_CONTENTS` to get a span of characters from the text subwindow. It gets the 1000 characters beginning at position 500 out of the text subwindow and places them into a null-terminated string.

```
#define TO_READ 1000

char          buf[TO_READ+1];
Textsw_index next_pos;

next_pos = (Textsw_index)
    window_get(textsw, TEXTSW_CONTENTS, 500, buf, TO_READ);

if (next_pos != 500+TO_READ) {
    Error case
} else
    buf[TO_READ] = '\0';
```


8.5. Editing the Contents of a Text Subwindow

The file or memory being edited by a text subwindow is referred to as the *backing store*. Several attributes and functions are provided to allow you to manipulate the backing store of a text subwindow.⁵⁰ This section describes the procedures and attributes that you can use to edit a text subwindow.

Removing Characters

You can remove a contiguous span of characters from a text subwindow by calling:

```
Textsw_index
textsw_delete(textsw, first, last_plus_one)
    Textsw      textsw;
    Textsw_index first, last_plus_one;
```

first specifies the first character of the span that will be deleted, while *last_plus_one* specifies the first character after the span that will not be deleted. *first* should be less than, or equal to, *last_plus_one*. To delete to the end of the text, pass the special value `TEXTSW_INFINITY` for *last_plus_one*.

The return value is the number of characters deleted, and is *last_plus_one* - *first*, unless all or part of the specified span is read-only. In this case, only those characters that are not read-only will be deleted, and the return value will indicate how many such characters there were. If the insertion point is in the span being deleted, it will be left at *first*.

A side-effect of calling `textsw_delete()` is that the deleted characters become the contents of the global Clipboard. To remove the characters from the textsw subwindow without affecting the Clipboard, call:

```
Textsw_index
textsw_erase(textsw, first, last_plus_one)
    Textsw      textsw;
    Textsw_index first, last_plus_one;
```

Again, the return value is the number of characters removed, and *last_plus_one* can be `TEXTSW_INFINITY`.

Both of these procedures will return 0 if the operation fails.

Emulating an Editing Character

You can emulate the behavior of an editing character, such as CTRL-H, with `textsw_edit()`:

```
Textsw_index
textsw_edit(textsw, unit, count, direction)
    Textsw      textsw;
    unsigned    unit, count, direction;
```

⁵⁰ Note that the edit log maintained by the text subwindow package is reset on each operation affecting the backing store. For a description of the edit log, see the discussion at the end of *Editing the Contents of a Text Subwindow*.

Depending on the value of `unit`, this routine will erase either a character, a word, or a line. Set `unit` to:

- `TEXTSW_UNIT_IS_CHAR` to erase individual characters,
- `TEXTSW_UNIT_IS_WORD` to erase the span of characters that make up a word (including any intervening white space or other non-word characters), or
- `TEXTSW_UNIT_IS_LINE` to erase all characters in the line on one side of the insertion point.

If the `direction` parameter is 0, the operation will affect characters after the insertion point, otherwise it will affect characters before the insertion point.

The number of times the operation will be applied is determined by the value of the `count` parameter. Set it to one to do the edit once, or to a value greater than one to do multiple edits in a single call. `textsw_edit()` returns the number of characters actually removed.

For example, suppose you want to interpret the function key `[F7]` as meaning “delete word forward”. On receiving the event code for the `[F7]` key going up, you would make the call:

```
textsw_edit(textsw, TEXTSW_UNIT_IS_WORD, 1, 0);
```

Replacing Characters

While a span of characters may be replaced by calling `textsw_erase()` followed by `textsw_insert()`, character replacement is done most efficiently by calling:

```
Textsw_index
textsw_replace_bytes(textsw, first, last_plus_one, buf, buf_len)
    Textsw      textsw;
    Textsw_index first, last_plus_one;
    char        *buf;
    int         buf_len;
```

The span of characters to be replaced is specified by `first` and `last_plus_one`, just as in the call to `textsw_erase()`. The new characters are specified by `buf` and `buf_len`, just as in the call to `textsw_insert()`. Once again, if `last_plus_one` is `TEXTSW_INFINITY`, the replace affects all characters from `first` to the end of the text. If the insertion point is in the span being replaced, it will be left at `first + buf_len`.

The return value is the net number of bytes inserted. The number is negative if the original string is longer than the one which replaces it. If a problem occurs when an attempt is made to replace a span, then it will return an error code of 0.

`textsw_replace_bytes()`, like `textsw_erase()`, does *not* put the characters it removes on the global Clipboard.

The Editing Log

All text subwindows allow the user to undo editing actions. In order to implement this feature, the text subwindow package keeps a running log of all the edits. If there is a file associated with the text subwindow, this log is kept in a file in the `/tmp` directory.

This file can grow until this directory runs out of space. To limit the size of the edit log and to avoid filling up all of `tmp` the user can set the text wrap around size in the `defaultsedit(1) Tty/text_wraparound_size`. If there is no associated file, the edit log is kept in memory, and the maximum size of the log is controlled by the attribute `TEXTSW_MEMORY_MAXIMUM`, which defaults to 20,000 bytes.

Unfortunately, once an edit log kept in memory has reached its maximum size, no more characters can be inserted into or removed from the text subwindow. In particular, since deletions, as well as insertions, are logged, space cannot be recovered by deleting characters.

It is important to understand how the edit log works because you may want to use a text subwindow with no associated file to implement a temporary scratch area or error message log. If such a text subwindow is used for a long time, the default limit of 20,000 bytes may well be reached, and it will be impossible for either the user or your code to insert any more characters even though there may be only a few characters visible in the text subwindow. Therefore, it is recommended to set `TEXTSW_MEMORY_MAXIMUM` much higher, say to 200,000.

Which File is Being Edited?

To find out which file the text subwindow is editing, call:

```
int
textsw_append_file_name(textsw, name)
    Textsw textsw;
    char *name;
```

If the text subwindow is editing memory, then this routine will return a non-zero value. Otherwise, it will return 0, and append the name of the file to the end of `name`.

Interactions with the File System

If a text subwindow is editing a file called *myfile* and the user chooses 'Save Current File' from the subwindow's menu (or client code invokes `textsw_save()`), the following sequence of file operations occurs:

- `myfile` is copied to `myfile%`
- The contents of `myfile%` are combined with information from the edit log file (`/tmp/TextProcess-id.Counter`) and written over `myfile` (thereby preserving all its permissions, etc).
- The edit log file is removed from `/tmp`.

If `myfile` is a symbolic link to `./some_dir/otherfile`, then the backup file is created as `./some_dir/otherfile%`.

Keep in mind that the user can change the current directory by selecting 'Load File' or 'Set Directory' from the text subwindow menu. If `myfile` is a relative path name, then the copy to `myfile%` and the save take place in the current directory.

8.6. Saving Edits in a Subwindow

To save any edits made to a file currently loaded into a text subwindow call:

```
unsigned
textsw_save(textsw, locx, locy)
    Textsw textsw;
    int     locx, locy;
```

`locx` and `locy` are relative to the upper left corner of the text subwindow and are used to position the upper left corner of the alert should the save fail for some reason — usually they should be 0. The return value is 0 if and only if the save succeeded.

Storing Edits

The text subwindow may not contain a file, or the client may wish to place the edited version of the text (whether or not the original text came from a file) in some specific file. To store the contents of a text subwindow to a file call:

```
unsigned
textsw_store_file(textsw, filename, locx, locy)
    Textsw textsw;
    char   *filename;
    int     locx, locy;
```

Again, `locx` and `locy` are used to position the upper left corner of the message box. The return value is 0 if and only if the store succeeded.

NOTE *By default, this call changes the file that the text subwindow is editing, so that subsequent saves will save the edits to the new file. To override this policy, set the attribute `TEXTSW_STORE_CHANGES_FILE` to `FALSE`.*

Discarding Edits

To discard the edits performed on the contents of a text subwindow, call:

```
void
textsw_reset(textsw, locx, locy)
    Textsw textsw;
    int     locx, locy;
```

`locx` and `locy` are as above. Note that if the text subwindow contains a file which has not been edited, the effect of `textsw_reset` is to unload the file and replace it by memory provided by the text subwindow package; thus the user will see an absolutely empty text subwindow. Alternatively, if the text subwindow already was editing memory then another, untouched, piece of primary memory will be provided and the edited piece will be deallocated.

The rest of this chapter describes the other functions that are available for text subwindows. These features include setting the contents of a subwindow, setting the primary selection, and how to deal with multiple or split views.

8.7. Setting the Contents of a Text Subwindow

You may want to set the initial contents of a text subwindow that your application uses. For example, the SunView `mailto` sets the initial contents of the composition window to come up with the headings To, Subject, and so on.

To set the initial contents of a text subwindow, use one of three attributes: `TEXTSW_INSERT_FROM_FILE`, `TEXTSW_FILE_CONTENTS`, and `TEXTSW_CONTENTS`. Each attribute is illustrated in code fragments given below.

`TEXTSW_FILE_CONTENTS`

The attribute `TEXTSW_FILE_CONTENTS` makes it possible for a client to initialize the text subwindow contents from a file yet still edit the contents in memory. The user can return a text subwindow to its initial state after an editing session by choosing 'Undo All Edits' in the text menu.

The following code fragment shows how you would use this attribute.

```
Textsw      textsw;
char        *filename;
Textsw_index pos;

window_set(textsw,
           TEXTSW_FILE_CONTENTS, filename,
           TEXTSW_FIRST, pos,
           0);
```

When the client calls the undo routine and `filename` is not a null string, then it will initialize the memory used by the text subwindow with the contents of the file specified by `filename`.

When the client calls the undo routine and the `filename` is a null string, then it will initialize the memory used by the text subwindow with the previous contents of the text subwindow.

`TEXTSW_CONTENTS`

`TEXTSW_CONTENTS` lets you insert a text string from memory, instead of a file, into the text subwindow. The default for this attribute is `NULL`.

If you use `window_create()` with this attribute, then it will specify the initial contents for a non-file text subwindow.

If you use `window_set()` with this attribute it will set the contents of a window as in:

```
window_set(textsw, TEXTSW_CONTENTS, "text", 0);
```

If you use `window_get()` with this attribute, then you will need to provide additional parameters as in:

```
window_get(textsw, TEXTSW_CONTENTS, pos, buf, buf_len)
```

The return value is the next position to be read. The buffer array

`buf[0..buf_len-1]` is filled with the characters from `textsw` beginning at the index `pos`, and is null-terminated only if there were too few characters to fill the buffer.

TEXTSW_INSERT_FROM_FILE `TEXTSW_INSERT_FROM_FILE` allows a client to insert the contents of a file into a text subwindow at the current insertion point. It is the programming equivalent of a user choosing 'Include File' from the text menu.

The following code fragment is a sample of using this attribute.

```
Textsw      textsw;
Textsw_status status;

window_set(textsw,
            TEXTSW_STATUS, &status,
            TEXTSW_INSERT_FROM_FILE, filename,
            0);
```

Three status values may be returned for this attribute when the argument `TEXTSW_STATUS` is passed in the same call to `window_create()` or `window_set()`:

- `TEXTSW_STATUS_OKAY` — the operation was successful.
- `TEXTSW_STATUS_CANNOT_INSERT_FROM_FILE` — the operation failed
- `TEXTSW_STATUS_OUT_OF_MEMORY` — the function cannot insert the text, because it ran out of memory

8.8. Positioning the Text Displayed in a Text Subwindow

Usually there is more text managed by the text subwindow than can be displayed all at once. As a result, it is often necessary to determine the indices of the characters that are being displayed, and to control exactly which portion of the text is being displayed.

Screen Lines and File Lines

When there are long lines in the text it is necessary to draw a distinction between two different definitions of "line of text."

A *screen line* reflects what is actually displayed on the screen. A line begins with the leftmost character in the subwindow and continues across until either a newline character or the right edge of the subwindow is encountered. A *file line*, on the other hand, can only be terminated by the newline character. It is defined as the span of characters starting after a newline character (or the beginning of the file) running through the next newline character (or the end of the file).

Whenever the right edge of the subwindow is encountered before the newline, if `TEXTSW_LINE_BREAK_ACTION` is `TEXTSW_WRAP_AT_CHAR`, the next character and its successors will be displayed on the next lower screen line. In this case there would be two screen lines, but only one file line. From the perspective of the display there are two lines; from the perspective of the file only one. If, on the other hand `TEXTSW_LINE_BREAK_ACTION` is `TEXTSW_WRAP_AT_WORD`, the entire word will be displayed on the next line.

Unless otherwise specified, all text subwindow attributes and procedures use the *file line* definition.

NOTE *Line indices have a zero-origin, like the character indices; that is, the first line has index 0, not 1.*

Absolute Positioning

Two attributes are provided to allow you to specify which portion of the text is displayed in the text subwindow.

Setting the attribute TEXTSW_FIRST to a given index causes the first character of the line containing the index to become the first character displayed in the text subwindow. Thus the following call causes the text to be positioned so that the first displayed character is the first character of the line which contains index 1000. This call only positions one view at a time:

```
window_set(textsw, TEXTSW_FIRST, 1000, 0);
```

To position all of the views in a text subwindow, use the attribute TEXTSW_FOR_ALL_VIEWS as in the following call:

```
window_set(textsw, TEXTSW_FOR_ALL_VIEWS, TRUE,
            TEXTSW_FIRST, 1000, 0);
```

Conversely, the following call retrieves the index of the first displayed character:

```
index = (Textsw_index>window_get(textsw, TEXTSW_FIRST);
```

A related attribute, useful in similar situations, is TEXTSW_FIRST_LINE. When used in a call on window_set() or window_get(), the value is a file line index within the text.

You can determine the character index that corresponds to a given line index (both zero-origin) within the text by calling:

```
Textsw_index
textsw_index_for_file_line(textsw, line)
    Textsw textsw;
    int    line;
```

The return value is the character index for the first character in the line, so character index 0 always corresponds to line index 0.

Relative Positioning

To move the text in a text subwindow up or down by a small number of lines, call the routine:

```
void
textsw_scroll_lines(textsw, count)
    Textsw textsw;
    int    count;
```

A positive value for count causes the text to scroll up, just as if the user had used the left mouse button in the scrollbar, while a negative value causes the text

How Many Screen Lines are in the Subwindow?

to scroll down, as if the user had used the right mouse button in the scrollbar.

When calling `textsw_scroll_lines()` you may want to know how many screen lines are in the text subwindow. You can find this out by calling:

```
int
textsw_screen_line_count(textsw)
    Textsw textsw;
```

Which File Lines are Visible?

Exactly which file lines are visible on the screen is determined by calling:

```
void
textsw_file_lines_visible(textsw, top, bottom)
    Textsw textsw;
    int *top, *bottom;
```

This routine fills in the addressed integers with the file line indices of the first and last file lines being displayed in the specified text subwindow.

Guaranteeing What is Visible

To ensure that a particular line or character is visible, call:

```
void
textsw_possibly_normalize(textsw, position)
    Textsw textsw;
    Textsw_index position;
```

The text subwindow must be displayed on the screen, before this function will work.

If the character at the specified `position` is already visible, then this routine does nothing. If it is not visible, then it repositions the text so that it is visible and at the top of the subwindow.

If a particular character should always be at the top of the subwindow, then calling the following routine is more appropriate:

```
void
textsw_normalize_view(textsw, position)
    Textsw textsw;
    Textsw_index position;
```

Ensuring that the Insertion Point is Visible

Most of the programmatic editing actions do not update the text subwindow to display the caret, even if `TEXTSW_INSERT_MAKES_VISIBLE` is set. If you want to ensure that the insertion point is visible, call something like

```
textsw_possibly_normalize(textsw,
    (Textsw_index) window_get(textsw, TEXTSW_INSERTION_POINT);
```


8.9. Finding and Matching a Pattern

A common operation performed on text is finding a span of characters that match some specification. The text subwindow provides several rudimentary pattern matching facilities. This section describes two functions that you may call in order to perform similar operations.

Matching a Span of Characters

To find the nearest span of characters that match a pattern, call:

```
int
textsw_find_bytes(textsw, first, last_plus_one, buf,
                  buf_len, flags)
    Textsw          textsw;
    Textsw_index    *first, *last_plus_one;
    char            *buf;
    unsigned        buf_len;
    unsigned        flags;
```

The pattern to match is specified by `buf` and `buf_len`. The matcher looks for an exact and literal match — it is sensitive to case, and does not recognize any kind of meta-character in the pattern. `first` specifies the position at which to start the search. If `flags` is 0, the search proceeds forwards through the text, if 1 the search proceeds backwards. The return value is -1 if the pattern cannot be found, else it is some non-negative value, in which case the indices addressed by `first` and `last_plus_one` will have been updated to indicate the span of characters that match the pattern.

Matching a Specific Pattern

Another useful operation is to find delimited text. For example, you might want to find the starting brace and the ending brace in a piece of code. To find a matching pattern, call:

```
int
textsw_match_bytes(textsw, first, last_plus_one,
                  start_sym, start_sym_len,
                  end_sym, end_sym_len, field_flag)
    Textsw          textsw;
    Textsw_index    *first, *last_plus_one;
    char            *start_sym, *end_sym;
    int             start_sym_len, end_sym_len;
    unsigned        field_flag;
```

`first` stores the starting position of the pattern that you want to search for. `last_plus_one` stores the cursor position of the end pattern. Its value is one position past the text. `start_sym` and `end_sym` store the beginning position and ending position of the pattern respectively. `start_sym_len` and `end_sym_len` store starting and ending pattern's length respectively.

Use one of the three field flag values to search for matches:

TEXTSW_DELIMITER_FORWARD, TEXTSW_DELIMITER_BACKWARD, and TEXTSW_DELIMITER_ENCLOSE.

- TEXTSW_DELIMITER_FORWARD begins from `first` and searches forward until it finds `start_sym` and matches it forward with `end_sym`.

- TEXTSW_DELIMITER_BACKWARD begins from `first` and searches backward for `end_sym` and matches it backward with `start_sym`.
- TEXTSW_DELIMITER_ENCLOSE begins from `first` and expands both directions to match `start_sym` and `end_sym` of the next level.

If no match is found, then `textsw_match_bytes()` will return a value of `-1`. If a match is found, then it will return the index of the first match.

The following code fragment is an example of finding delimited text. Notice that the `field_flag` value is `TEXTSW_DELIMITER_FORWARD`.

```
Textsw_index    first, last_plus_one, pos;

first = (Textsw_index) window_get(textsw, TEXTSW_INSERTION_POINT);
pos = textsw_match_bytes(textsw, &first, &last_plus_one,
                        "/*", strlen("/*/"),
                        "/*", strlen("/*/"), TEXTSW_DELIMITER_FORWARD);
if (pos > 0) {
    textsw_set_selection(textsw, first, last_plus_one, 1);
    window_set(textsw, TEXTSW_INSERTION_POINT, last_plus_one, 0);
} else
    (void) window_bell(textsw);
```

This code searches forward from `first` until it finds the starting `/*` and matches it forward with the next `/*`. If no match is found, a bell will ring in the text subwindow.

8.10. Marking Positions

Often a client wants to keep track of a particular character, or group of characters that are in the text subwindow. Given that arbitrary editing can occur in a text subwindow, and that it is very tedious to intercept and track all of the editing operations applied to a text subwindow, it is often easier to simply place one or more marks at various positions in the text subwindow. These marks are automatically updated by the text subwindow to account for user and client edits. There is no limit to the number of marks you can add.

A new mark is created by calling:

```
Textsw_mark
textsw_add_mark(textsw, position, flags)
    Textsw      textsw;
    Textsw_index position;
    unsigned    flags;
```

The `flags` argument is either `TEXTSW_MARK_DEFAULTS` or `TEXTSW_MARK_MOVE_AT_INSERT`. The latter causes an insertion that occurs at the marked position to move the mark to the end of the inserted text, whereas the former causes the mark to not move when text is inserted at the mark's current position. As an example, suppose that the text managed by the text subwindow consists of the two lines

```
this is the first line
not this, which is the second
```


Assume a mark is set at position 5 (just before the *i* in *is* on the first line) with flags of TEXTSW_MARK_MOVE_AT_INSERT.

When the user selects just before the *is* (thereby placing the insertion point before the *i*, at position 5) and types an *Bh*, making the text read

```
this his the first line
not this, which is the second
```

the mark moves with the insertion point and they both end up at position 6.

However, if the flags had been TEXTSW_MARK_DEFAULTS, then the mark would remain at position 5 after the user typed the *h*, although the insertion point moved on to position 6.

Now, suppose instead that the user had selected before the *this* on the first line, and typed **Kep**, making the text read

```
Kepthis is the first line
not this, which is the second
```

In this case, no matter what flags the mark had been created with, it would end up at position 8, still just before the *i* in *is*.

If a mark is in the middle of a span of characters that is subsequently deleted, the mark moves to the beginning of the span. Going back to the original scenario, with the original text and the mark set at position 5, assume that the user deletes from the *h* in *this* through the *e* in *the* on the first line, resulting in the text

```
te first line
not this, which is the second
```

When the user is done, the mark will be at position 1, just before the *e* in *te*.

The current position of a mark is determined by calling:

```
Textsw_index
textsw_find_mark(textsw, mark)
    Textsw      textsw;
    Textsw_mark mark;
```

An existing mark is removed by calling:

```
void
textsw_remove_mark(textsw, mark)
    Textsw      textsw;
    Textsw_mark mark;
```

Note that marks are dynamically allocated, and it is the client's responsibility to keep track of them and to remove them when they are no longer needed.

8.11. Setting the Primary Selection

The primary selection may be set by calling:

```
void
textsw_set_selection(textsw, first, last_plus_one, type)
    Textsw      textsw;
    Textsw_index first, last_plus_one;
    unsigned    type;
```

A value of 1 for type means *primary selection*, while a value of 2 means *secondary selection*, and a value of 17 is *pending delete*. Note that there is no requirement that all or part of the selection be visible; use

`textsw_possibly_normalize()` (described previously in Section 8.5, *Editing the Contents of a Text Subwindow*) to guarantee visibility.

8.12. Dealing with Multiple Views

By using the 'Split View' menu operation, the user can create multiple views of the text being managed by the text subwindow. Although these additional views are usually transparent to the client code controlling the text subwindow, it may occasionally be necessary for a client to deal directly with all of the views. This is accomplished by using the following routines, and the information that split views are simply extra text subwindows that happen to share the text of the original text subwindow.

```
Textsw
textsw_first(textsw)
    Textsw textsw;
```

Given an arbitrary view out of a set of multiple views, `textsw_first()` returns the first view (currently, this is the original text subwindow that the client created). To move through the other views of the set, call:

```
Textsw
textsw_next(textsw)
    Textsw textsw;
```

Given any view of the set, `textsw_next()` returns some other member of the set, or NULL if there are none left to enumerate. The following loop is guaranteed to process all of the views in the set:

```
for (textsw=textsw_first(any_split);
    *textsw;
    textsw=textsw_next(textsw)) {
    processing_involving textsw;
}
```

When you create a text subwindow take into account that your user may split the window. If you do something like try to enlarge the window, you will run into problems.

8.13. Notifications from a Text Subwindow

The text subwindow notifies its client about interesting changes in the subwindow's or text's state by calling a notification procedure. It also calls this procedure in response to user actions. If the client does not provide an explicit notification procedure by using the attribute `TEXTSW_NOTIFY_PROC`, then the text subwindow provides a default procedure. The declaration for this procedure looks like:

```
void
notify_proc(textsw, avlist)
    Textsw      textsw;
    Attr_avlist avlist;
```

`avlist` contains attributes that are the members of the `Textsw_action` enumeration.

Your notification procedure must be careful to either process all of the possible attributes that it can be called with or to pass through the attributes that it does not process to the standard notification procedure. This is important because among the attributes that can be in the *avlist* are those that cause the standard notification procedure to implement the *Front*, *Back*, *Open*, *Close*, and *Quit* accelerators of the user interface.

Here is an example of a client notify procedure, and a code fragment demonstrating how it would be used:

```
int (*default_textsw_notify)();

void
client_notify_proc(textsw, attributes)
    Textsw      textsw;
    Attr_avlist attributes;
{
    int pass_on = FALSE;
    Attr_avlist attrs;

    for (attrs = attributes; *attrs;
         attrs = attr_next(attrs)) {
        switch ((Textsw_action)(*attrs)) {
            case TEXTSW_ACTION_CAPS_LOCK:
                /* Swallow this attribute */
                ATTR_CONSUME(*attrs);
                break;
            case TEXTSW_ACTION_CHANGED_DIRECTORY:
                /* Monitor the attribute, don't swallow it */
                strcpy(current_directory, (char *)attrs[1]);
                pass_on = TRUE;
                break;
            default:
                pass_on = TRUE;
                break;
        }
    }
    if (pass_on)
        (void) default_textsw_notify(textsw, attributes);
}
```

```
default_textsw_notify =
(void (*)( ))window_get(textsw, TEXTSW_NOTIFY_PROC);
window_set(textsw, TEXTSW_NOTIFY_PROC, client_notify_proc);
```

The `Textsw_action` attributes which may be passed to your notify procedure are listed in the following table (duplicated in Chapter 19, *SunView Interface Summary*). Remember that they constitute a special class of attributes which are passed to your textsw notification procedure. They are not attributes of the text subwindow in the usual sense, and can not be retrieved or modified using `window_get()` or `window_set()`.

Table 8-2 *Textsw_action Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
TEXTSW_ACTION_CAPS_LOCK	boolean	The user pressed the CAPS-lock function key to change the setting of the CAPS-lock (it is initially 0, meaning off).
TEXTSW_ACTION_CHANGED_DIRECTORY	char *	The current working directory for the process has been changed to the directory named by the provided string value.
TEXTSW_ACTION_EDITED_FILE	char *	The file named by the provided string value has been edited. Appears once per session of edits (see below).
TEXTSW_ACTION_EDITED_MEMORY	none	monitors whether an empty text subwindow has been edited.
TEXTSW_ACTION_FILE_IS_READONLY	char *	The file named by the provided string value does not have write permission.
TEXTSW_ACTION_LOADED_FILE	char *	The text subwindow is being used to view the file named by the provided string value.
TEXTSW_ACTION_TOOL_CLOSE	(no value)	The frame containing the text subwindow should become iconic.
TEXTSW_ACTION_TOOL_DESTROY	Event *	The tool containing the text subwindow should exit, without checking for a veto from other subwindows. The value is the user action that caused the destroy.
TEXTSW_ACTION_TOOL_QUIT	Event *	The tool containing the text subwindow should exit normally. The value is the user action that caused the exit.
TEXTSW_ACTION_TOOL_MGR	Event *	The tool containing the text subwindow should do the window manager operation associated with the provided event value.
TEXTSW_ACTION_USING_MEMORY	(no value)	The text subwindow is being used to edit a string stored in primary memory, not a file.

The attribute `TEXTSW_ACTION_EDITED_FILE` is a slight misnomer, as it is given to the notify procedure *after* the first edit to *any* text, whether or not it came from a file. This notification only happens once per session of edits, where notification of `TEXTSW_ACTION_LOADED_FILE` is considered to terminate the old session and start a new one.

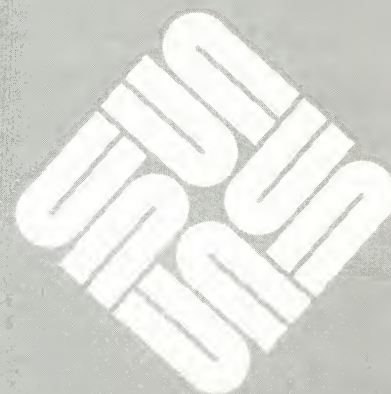
NOTE *The attribute `TEXTSW_ACTION_LOADED_FILE` must be treated very carefully.* This is because the notify procedure gets called with this attribute in several situations: after a file is initially loaded, after any successful 'Save Current File' menu operation, after a 'Undo All Edits' menu operation, and during successful calls to `textsw_reset()`, `textsw_save()` and `textsw_store()`.

The appropriate response by the procedure is to interpret these notifications as being equivalent to:

"The text subwindow is displaying the file named by the provided string value; no edits have been performed on the file yet. In addition, any previously displayed or edited file has been either reset, saved, or stored under another name."

Panels

Panels	153
9.1. Introduction to Panels and Panel Items	158
Message Items	158
Button Items	158
Choice Items	158
Toggle Items	159
Text Items	159
Slider Items	159
9.2. Basic Panel Routines	159
Creating and Sizing Panels	159
Creating and Positioning Panel Items	160
Explicit Item Positioning	160
Default Item Positioning	161
Laying Out Components Within an Item	162
Modifying Attributes	162
Panel-Wide Item Attributes	163
Retrieving Attributes	163
Destroying Panel Items	164
9.3. Using Scrollbars With Panels	165
Creating Scrollbars	165
Scrolling Panels Which Change Size	165
Detaching Scrollbars from Panels	166
9.4. Messages	167



9.5. Buttons	167
Button Selection	167
Button Notification	167
Button Image Creation Utility	168
9.6. Choices	170
Displaying Choice Items	170
Choice Selection	172
Choice Notification	172
Choice Value	172
Choice Menus	172
9.7. Toggles	176
Displaying Toggles	176
Toggle Selection	176
Toggle Notification	176
Toggle Value	176
Toggle Menus	178
9.8. Text	178
Displaying Text Items	178
Text Selection	179
Text Notification	180
Writing Your Own Notify Procedure	181
Text Value	182
Text Menus	183
9.9. Sliders	184
Displaying Sliders	184
Slider Selection	184
Slider Notification	184
Slider Value	185
9.10. Painting Panels and Individual Items	185
9.11. Iterating Over a Panel's Items	188
9.12. Panel Item Client Data	188
9.13. Event Handling	189
Default Event Handling	189
Writing Your Own Event Handler	189
Translating Events from Panel to Window Space	193

Panels

This chapter describes the panel subwindow package, which you can use by including the file `<suntool/panel.h>`.

Section 1 provides a non-technical introduction to panels. Section 2 introduces the basic concepts and routines needed to use panels. Scrollable panels are covered in Section 3. Sections 4 through 9 describe the different types of panel items in detail, including examples.

For examples of complete panels, see the programs *filer*, *image_browser_1* and *image_browser_2*, which are listed in Appendix A and discussed in Chapter 4, *Using Windows*.

Quick Reference, Listings and Summary Tables

For quick reference, the next two pages are a visual index to the different effects possible in panels. After that come lists of the available panel and panel item attributes, functions and macros. Many of these are discussed in the rest of this chapter and elsewhere (use the *Index* to check). Finally, tables that summarize the usage of panel attributes, functions and macros are in Chapter 19, *SunView Interface Summary*:

- the Panel Attributes table begins on page 346;
- the *Generic Panel Item Attributes* table begins on page 347;
- the *Choice and Toggle Item Attributes* table begins on page 349;
- the *Slider Attributes* table begins on page 351;
- the *Text Item Attributes* table begins on page 352;
- the *Panel Functions and Macros* table begins on page 353.

Page Description

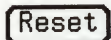
Example

167 Messages



This action will cause unsaved edits to be lost.

168 Buttons



173 Choice (default)

Drawing Mode: ☐ Points ☒ Line ☐ Rectangle ☐ Circle ☐ Text

173 Choice (custom marks)

Drawing Mode: Points ☒ Line ☐ Rectangle ☐ Circle ☐ Text

173 Choice (inverted)

Drawing Mode: Points ☒ Line ☐ Rectangle ☐ Circle ☐ Text

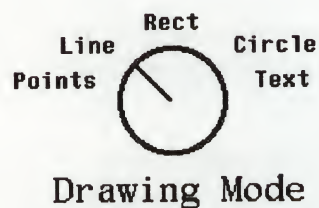
174 Choice (current)

Drawing Mode: Line

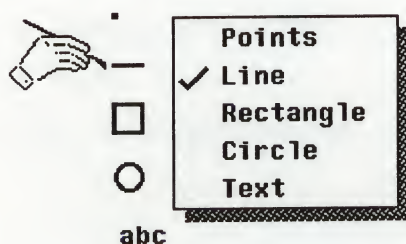
174 Choice (cycle)

Drawing Mode: ☒ Line

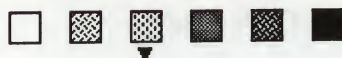
175 Choice (dial)



175 Choice (images, menu)



171 Choice (images)



Page Description

Example

177 Toggle (vertical)

Format Options:

- ☒ Long
☐ Reverse
☒ Show all files

178 Text

Name: Edward G. Robinson

179 Text (masked)

Password: *****

183 Text with menu

File: dervish.image

- ESC - Filename completion
- ^L - Load image from file
- ^S - Store image to file
- ^B - Browse directory
- ^Q - Quit

185 Slider

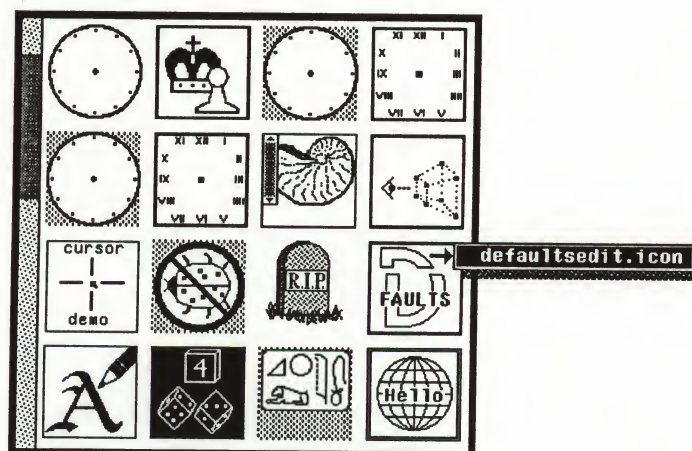
Brightness: [75] 0  100

168 Button with menu

Introduction Pixwins Cursors
 SunView Model Text Subwindows Icons
 Windows → Panels Scrollbars
 Canvases TTY Subwindows Selection Service
 Input Menus Notifier

SunView Manual

194 Buttons with menus on scrollable panel



Panel Attributes

PANEL_ACCEPT_KEYSTROKE	PANEL_EVENT_PROC	PANEL_LABEL_BOLD
PANEL_BACKGROUND_PROC	PANEL_FIRST_ITEM	PANEL_LAYOUT
PANEL_BLINK_CARET	PANEL_ITEM_X_GAP	PANEL_SHOW_MENU
PANEL_CARET_ITEM	PANEL_ITEM_Y_GAP	

Generic Panel Item Attributes

PANEL_ACCEPT_KEYSTROKE	PANEL_MENU_CHOICE_IMAGES
PANEL_CLIENT_DATA	PANEL_MENU_CHOICE_STRINGS
PANEL_EVENT_PROC	PANEL_MENU_CHOICE_VALUES
PANEL_ITEM_RECT	PANEL_MENU_TITLE_FONT
PANEL_ITEM_X	PANEL_MENU_TITLE_IMAGE
PANEL_ITEM_Y	PANEL_MENU_TITLE_STRING
PANEL_LABEL_X	PANEL_NEXT_ITEM
PANEL_LABEL_Y	PANEL_NOTIFY_PROC
PANEL_LABEL_BOLD	PANEL_PAINT
PANEL_LABEL_FONT	PANEL_PARENT_PANEL
PANEL_LABEL_IMAGE	PANEL_SHOW_ITEM
PANEL_LABEL_STRING	PANEL_SHOW_MENU
PANEL_LAYOUT	PANEL_VALUE_X
PANEL_MENU_CHOICE_FONTS	PANEL_VALUE_Y

Choice and Toggle Item Attributes

PANEL_CHOICE_FONTS	PANEL_MARK_IMAGE
PANEL_CHOICE_IMAGE	PANEL_MARK_IMAGES
PANEL_CHOICE_IMAGES	PANEL_MARK_X
PANEL_CHOICE_STRING	PANEL_MARK_XS
PANEL_CHOICE_STRINGS	PANEL_MARK_Y
PANEL_CHOICE_X	PANEL_MARK_YS
PANEL_CHOICE_XS	PANEL_MENU_MARK_IMAGE
PANEL_CHOICE_Y	PANEL_MENU_NOMARK_IMAGE
PANEL_CHOICE_YS	PANEL_NOMARK_IMAGE
PANEL_CHOICES_BOLD	PANEL_NOMARK_IMAGES
PANEL_DISPLAY_LEVEL	PANEL_SHOW_MENU_MARK
PANEL_FEEDBACK	PANEL_TOGGLE_VALUE
PANEL_LAYOUT	PANEL_VALUE

Slider Item Attributes

PANEL_MIN_VALUE	PANEL_SHOW_RANGE	PANEL_VALUE
PANEL_MAX_VALUE	PANEL_SHOW_VALUE	PANEL_VALUE_FONT
PANEL_NOTIFY_LEVEL	PANEL_SLIDER_WIDTH	

Text Item Attributes

PANEL_MASK_CHAR	PANEL_VALUE_DISPLAY_LENGTH
PANEL_NOTIFY_LEVEL	PANEL_VALUE
PANEL_NOTIFY_STRING	PANEL_VALUE_FONT
PANEL_VALUE_STORED_LENGTH	

Panel Functions and Macros

```

panel_accept_key(object, event)
panel_accept_menu(object, event)
panel_accept_preview(object, event)
panel_advance_caret(panel)
panel_backup_caret(panel)
panel_begin_preview(object, event)
panel_button_image(panel, string, width, font)
panel_cancel_preview(object, event)
panel_create_item(panel, item_type, attributes)
panel_default_handle_event(object, event)
panel_destroy_item(item)
panel_each_item(panel, item)
panel_event(panel, event)
panel_get(item, attribute[, optional_arg])
panel_get_value(item)
panel_paint(panel_object, paint_behavior)
panel_set(item, attributes)
panel_set_value(item, value)
panel_text_notify(item, event)
panel_update_preview(object, event)
panel_update_scrolling_size(panel)
panel_window_event(panel, event)

```


9.1. Introduction to Panels and Panel Items

Panels contain *items* through which the user interacts with a program. Panels are quite flexible; you can use them to model a variety of things, including:

- a form consisting mainly of text items;
- a message window containing status or error messages;
- a complex control panel containing items and menus of many types.

Panels need not be limited to the size of the subwindow they appear in. By attaching *scrollbars* to a panel, you can show a large panel within a smaller subwindow. The user can then bring the area of interest into view by means of the scrollbars.

There are six basic types of panel items: messages, buttons, choices, toggles, text and sliders. Items are made up of one or more displayable components. One component shared by all item types is the *label*. An item label is either a string or a graphic image (i.e., a pointer to a *pixrect*). Button, choice, toggle, and text items also have a menu component. Thus the user may interact with most items in either of two ways: by selecting the item directly or by selecting from the item's menu.

Each item type is introduced briefly below.

Message Items

The only visible component of a message item is a label, which may be an image or a string in a specified font. Message items are useful for annotations of all kinds, including titles, comments, descriptions, pictures, and dynamic status messages.

Message items are selectable, and you may specify a *notify procedure* to be called when the item is selected.

Button Items

Button items allow the user of a program to initiate commands. Buttons, like message items, have a label, are selectable, and have a notify procedure. Button items differ from message items in that they have visible feedback for previewing and accepting the selection.

Choice Items

Choice items allow the user to select one choice from a list. The displayed form of a choice item can vary radically, depending on how you set its attributes. A choice item can be presented as:

- a horizontal or vertical list of choices, with all choices visible and the current choice indicated by a mark (such as a checkmark);
- a horizontal or vertical list of choices, with all choices visible and the current choice in reverse-video;
- a "cycle item", or list of choices with only the current choice visible. Selecting the item causes the next choice in the list to be selected and displayed;
- a dial, knob or switch with a pointer of some sort which turns to indicate one of several choices;
- a place holder for a pop-up menu, with only the label visible until the menu button is pressed.

Behind this flexibility of presentation lies a uniform structure consisting of a label, a list of choices, and, optionally, a corresponding lists of *on-marks* and *off-marks* used to indicate which choice is currently selected.

Toggle Items

In appearance and structure, toggle items are identical to choice items. The difference lies in the behavior of the two types of items when selected. In a choice item exactly one element of the list is selected, or *current*, at a time. A toggle item, on the other hand, is best understood as a list of elements which behave as toggles: each choice may be either on or off, independently of the other choices. Selecting a choice causes it to change state. There is no concept of a single current choice; at any given time all, some, or none of the choices may be selected.

Text Items

Text items are basically type-in fields with optional labels and menus. You can specify that your notify procedure be called on each character typed in, only on specified characters, or not at all. This allows an application such as a forms-entry program to process input on a per character, per field, or per screen basis.

Slider Items

Slider items allow the graphical representation and selection of a value within a range. They are appropriate for situations where it is desired to make fine adjustments over a continuous range of values. A familiar model would be a horizontal volume control lever on a stereo panel.

9.2. Basic Panel Routines

This section covers basic panel usage, including creating and sizing panels, creating and positioning panel items, modifying and retrieving the attributes for panels and panel items, and destroying panel items.

Creating and Sizing Panels

Like all windows in SunView, panels are created by calling the window creation routine with the appropriate type parameter:

```
Panel panel;
panel = window_create(frame, PANEL, 0);
```

The above call will produce a panel which extends to the bottom and right edges of the frame. You can specify the panel's dimensions explicitly in character units via `WIN_COLUMNS` and `WIN_ROWS`, or in pixel units via `WIN_WIDTH` and `WIN_HEIGHT`.⁵¹

⁵¹ For a fuller discussion of subwindow sizing and layout see are in Chapter 4, *Using Windows*.

Often you want the panel to be just high enough to encompass all of the items within it. After creating all of the items, *and before creating any other subwindows* in the frame, set the height of the panel by calling the macro `window_fit_height()`. This macro will compute the lowest point occupied by any of the panel's items and set the panel height to that point plus a bottom margin of four pixels. The macros `window_fit_width()` to set the width, and `window_fit()` to set both the height and the width, are also provided.

Creating and Positioning Panel Items

To create a panel item, call:

```
Panel_item
panel_create_item(panel, item_type, attributes)
    Panel        panel;
    <item type>   item_type;
    <attribute-list> attributes;
```

Values for `item_type` must be one of `PANEL_MESSAGE`, `PANEL_BUTTON`, `PANEL_CHOICE`, `PANEL_CYCLE`, `PANEL_TOGGLE`, `PANEL_TEXT`, or `PANEL_SLIDER`.

Explicit Item Positioning

The position of items within the panel may be specified explicitly by means of the attributes `PANEL_ITEM_X` and `PANEL_ITEM_Y`.⁵² `PANEL_ITEM_X` sets the left edge of the item's rectangle (the rectangle which encloses the item's label and value). `PANEL_ITEM_Y` sets the top edge of the item's rectangle.

All coordinate specification attributes interpret their values in pixel units. For simple panels and forms which do not make heavy use of images and have only one text font, it is usually more convenient to specify positions in character units — columns and rows rather than x's and y's. You can specify positions in character units with the `ATTR_ROW()` and `ATTR_COL()` macros,⁵³ which interpret their arguments as rows or columns, respectively, and convert the value to the corresponding number of pixels, based on the panel's font, as specified by `WIN_FONT`. Compare the two calls below:

```
panel_create_item(panel, PANEL_MESSAGE,
                  PANEL_LABEL_STRING, "Hi!",
                  PANEL_ITEM_X,      10,
                  PANEL_ITEM_Y,      20,
                  0);
```

⁵² Many attributes, such as those relating to item positioning, apply across all of the item types; these are called *generic* attributes. A comprehensive summary of these generic attributes is given in the *Generic Item Attributes* table in Chapter 19, *SunView Interface Summary*.

⁵³ `ATTR_ROW()` and `ATTR_COL()` are described fully in Chapter 18, *Attribute Utilities*.

```

panel_create_item(panel, PANEL_MESSAGE,
                  PANEL_LABEL_STRING, "Hi!",
                  PANEL_ITEM_X,       ATTR_COL(10),
                  PANEL_ITEM_Y,       ATTR_ROW(20),
                  0);

```

The first will place the item at pixel location (10,20), while the second will place the item at row 20, column 10.

NOTE *The value computed for ATTR_ROW() includes the top margin, given by WIN_TOP_MARGIN, and the value computed for ATTR_COL() includes the left margin, given by WIN_LEFT_MARGIN. The alternate macros ATTR_ROWS() and ATTR_COLS() are also provided, which compute values that do not include the margins.*

Default Item Positioning

If you create an item without specifying its position, it is placed just to the right of the item on the "lowest row" of the panel, where lowest row is defined as the maximum y-coordinate (PANEL_ITEM_Y) of all the items. So in the absence of specific instructions, items will be placed within the panel in *reading order* as they are created: beginning four pixels in from the left and four pixels down from the top, items are located from left to right, top to bottom. If an item will not fit on a row, and more of the item would be visible on the next row, it will be placed on the next row. The number of pixels left blank between items on a row may be specified by PANEL_ITEM_X_GAP, which has a default value of 10. The number of pixels left blank between rows of items may be specified by PANEL_ITEM_Y_GAP, which has a default value of 5.

The default position for the next item is computed after an item is created. But if a client calls panel_set() after creating an item in such a way that the enclosing rectangle of the item is altered, the default position for the next item will *not* be recomputed. So, for example,

```

item = panel_create_item(panel, PANEL_MESSAGE, 0);
panel_set(item, PANEL_LABEL_STRING, "Hi", 0);

item1 = panel_create_item(panel, PANEL_MESSAGE,
                           PANEL_LABEL_STRING, "There",
                           0);

```

will result in **There** overlapping **Hi**.

CAUTION

Choice items currently have problems with item "creep." Each time the label of a choice item is set, the position of the item will be evaluated. If the value's position has not been fixed (with VALUE_X/Y), the value is positioned after the label. The problem is that the label is baseline-adjusted for a choice item. If the item position is not given when the label is set, the choice item will creep down because of the baseline adjustment.

Laying Out Components Within an Item

You may also specify the layout of the various components within an item, by means of the attributes `PANEL_LABEL_X`, `PANEL_LABEL_Y`, `PANEL_VALUE_X`, `PANEL_VALUE_Y`, etc. If the components are not explicitly positioned, then the value is placed either eight pixels to the right of the label, if `PANEL_LAYOUT` is `PANEL_HORIZONTAL` (the default), or four pixels below the label, if `PANEL_LAYOUT` is `PANEL_VERTICAL`.

Modifying Attributes

This section describes how to modify the values of attributes of panels or individual panel items which have already been created.

Since panels are a type of window, their attributes are set with `window_set()`. To set attributes of panel items, use:

```
panel_set(item, attributes)
Panel_item item;
<attribute-list> attributes;
```

A macro is provided to ease the syntax for the common operation of setting an item's value (attribute `PANEL_VALUE`):

```
panel_set_value(item, value)
Panel_item item;
caddr_t value;
```

Several examples of setting attributes are given here; for a complete list of the attributes applying to panels and items, see the tables in are in Chapter 19, *SunView Interface Summary*.

To move a panel's caret to the text item `name_item`:

```
window_set(panel,
PANEL_CARET_ITEM,
name_item, 0);
```

To set the value of the choice item `format_item` to the third choice (choices are zero-based):

```
Panel_item format_item;
panel_set_value(format_item, 2);
```

The first call below creates a message which is initially "hidden" (not displayed on the screen); the second call displays the message:

```
warning = panel_create_item(panel, PANEL_MESSAGE,
PANEL_LABEL_STRING, "Warning: file will be deleted.",
PANEL_SHOW_ITEM, FALSE,
0);
```

...

```
panel_set(warning, PANEL_SHOW_ITEM, TRUE, 0);
```

NOTE *The values for string-valued attributes are dynamically allocated when they are set (at creation time or later). If a previous value was present, it is freed after the new string is allocated. This is in contrast to the storage-allocation policy for retrieving attributes, described in the section titled *Retrieving Attributes*.*

Panel-Wide Item Attributes

Some attributes which apply to items may be set for all items in the panel by setting them when the panel is created. Such attributes include whether items have menus, whether item labels appear in bold, whether items are laid out vertically or horizontally, and whether items are automatically repainted when their attributes are modified.⁵⁴ For example, the call:

```
panel = window_create(frame, PANEL
                        PANEL_SHOW_MENU, FALSE,
                        PANEL_LABEL_BOLD, TRUE,
                        PANEL_LAYOUT, PANEL_VERTICAL,
                        PANEL_PAINT, PANEL_NONE,
                        0);
```

overrides the defaults for all the attributes mentioned: any items subsequently created in that panel will not have menus, will have their labels printed in bold and their components laid out vertically, and will not be repainted automatically when their attributes are modified.

NOTE *When you set the attribute `PANEL_LAYOUT`, it will only affect the components in each item, not the items themselves. That is, all items in a panel will not be laid out vertically.*

Keep in mind that the panel-wide item attributes mentioned above are only used to supply default values for items which are subsequently created. This means, for example, that you cannot change all the item labels to bold by first creating the items and then setting `PANEL_LABEL_BOLD` to `TRUE` for the panel.

Retrieving Attributes

Use `window_get()` to retrieve attributes for a panel. To retrieve attributes applying to panel items, use:

```
caddr_t
panel_get(item, attribute[, optional_arg])
    Panel_item      item;
    Panel_attribute attribute;
    Panel_attribute optional_arg;
```

A macro is provided to ease the syntax for the common operation of getting an item's value (attribute `PANEL_VALUE`):

⁵⁴ For a complete list of panel-wide item attributes, see the *Panel Attributes* table in *SunView Interface Summary*.


```

caddr_t
panel_get_value(item, value)
    Panel_item item;
    caddr_t value;

```

Since the *_get() routines are used to retrieve attributes of all types, you should coerce the value returned into the type appropriate to the attribute being retrieved, as in the examples below.

To find out whether an item is currently being displayed on the screen:

```

int displayed;
displayed = (int)panel_get(item, PANEL_SHOW_ITEM);

```

To find out whether the caret in a panel is blinking or non-blinking:

```

int blinking;
blinking = (int>window_get(panel, PANEL_BLINK_CARET);

```

To get the image for a choice item's third (counting from zero) choice:

```

Pixrect *image;
image = (Pixrect *)panel_get(item, PANEL_CHOICE_IMAGE, 2);

```

The above example illustrates the use of the optional_arg argument, which is used for only a few item attributes.

NOTE panel_get() *does not dynamically allocate storage for the values it returns.* If the value returned is a pointer, it points directly into the panel's private data. It is your responsibility to copy the information pointed to. The policy for setting attributes is different: the values for string-valued attributes are dynamically allocated (see the note above under *Modifying Attributes*).

Destroying Panel Items

To destroy a panel item (and free its associated dynamic storage), call:

```

panel_destroy_item(item);
    Panel_item item;

```

9.3. Using Scrollbars With Panels

Creating Scrollbars

A *scrollable* panel is a large panel which can be viewed through a smaller subwindow by means of scrollbars.

Scrollbars come in two orientations: vertical and horizontal. The call below creates a panel with both vertical and horizontal scrollbars (as would be desirable in a long, many-columned table, for example):

```
panel = window_create(frame, PANEL,
                      WIN_VERTICAL_SCROLLBAR, scrollbar_create(0),
                      WIN_HORIZONTAL_SCROLLBAR, scrollbar_create(0),
                      0);
```

The values of the attributes `WIN_VERTICAL_SCROLLBAR` and `WIN_HORIZONTAL_SCROLLBAR` are the scrollbars which are returned by the `scrollbar_create()` calls.⁵⁵

Commonly the scrollbar will remain attached to the panel for the duration of the panel's existence, and there will be no need to modify the scrollbar's attributes. In this simple case, there is no need to save the handle returned by `scrollbar_create()`. If you desire to destroy the scrollbar, modify its attributes, or detach it from one panel and attach it to another, you must either save the handle or retrieve it from the panel.⁵⁶ For example, to destroy a panel's vertical scrollbar:

```
scrollbar_destroy(panel_get(panel, WIN_VERTICAL_SCROLLBAR));
panel_set(panel, WIN_VERTICAL_SCROLLBAR, NULL, 0);
```

Scrolling Panels Which Change Size

Often panels are used to display information for browsing. `iconedit(1)`, for example, uses a popup panel to allow the user to browse through the images in a directory. The easiest way to do this is to create the panel items anew each time different information is displayed. For example, the `iconedit` function which fills the browsing panel first destroys any existing panel items, then creates an item for each image found.

If you are going to change the size of the panel in this way, you must inform the scrollbar of the new size by calling the function:

```
panel_update_scrolling_size(panel)
Panel panel;
```

⁵⁵ The call `scrollbar_create(0)` produces a default scrollbar. It is usually best to create a default scrollbar and let the user specify how it looks via *defaultsedit*. You can, of course, override the user's default settings by explicitly setting the scrollbar's attributes. For a complete list of scrollbar attributes see Chapter 19, *SunView Interface Summary*.

⁵⁶ In order to save the scrollbar's handle or reference any scrollbar attributes you must include the file `<suntool/scrollbar.h>`.

The correct time to call `panel_update_scrolling_size()` is after you have created all the items and given them labels. If you don't update the scrollbar's idea of the panel's size, the size of the scrollbar's bubble will be wrong.

Detaching Scrollbars from Panels

You may want the same panel to be scrollable at one time, and not scrollable at another. The code fragment below illustrates how this can be accomplished by attaching and detaching a scrollbar from a panel:

```
panel = window_create(frame, PANEL, 0);
...
(create items, do any other processing...)
...
/* create scrollbar and attach it to panel */
sb = scrollbar_create(0);
panel_set(panel, WIN_VERTICAL_SCROLLBAR, sb, 0);
...
(panel functions with scrollbar...)
...
/* now detach scrollbar from panel */
panel_set(panel, WIN_VERTICAL_SCROLLBAR, NULL, 0);
...
(panel functions without scrollbar...)
...
scrollbar_destroy(sb);
```

Note that the two packages are to be considered from the application's viewpoint as independent packages which can be used together. The application, *not* the panel package, has the responsibility for creating any scrollbars. In order to free the application of the responsibility for destroying the scrollbar, panels, when they are destroyed, destroy any scrollbars attached to them. However, detaching a scrollbar from a panel, as in the above example, does not cause that scrollbar to be destroyed. The same scrollbar may be attached and detached from any number of panels any number of times.

The sections which follow discuss the six item types in detail.

9.4. Messages

Messages are the simplest of the item types. Their only visible component is their label. They have no value or menu.

Message items, like buttons, are selectable and can have notify procedures. The selection behavior of messages differs from that of buttons in that no feedback is given to the user when a message is selected.

Example

In the following example, two message items are used together to give a warning message:



This action will cause unsaved edits to be lost.

```
static short    stop_array[] = {
#include "stopsign.image"
};
mpr_static(stopsign, 64, 64, 1, stop_array);

panel_create_item(panel, PANEL_MESSAGE,
                  PANEL_LABEL_IMAGE, &stopsign,
                  0);

panel_create_item(panel, PANEL_MESSAGE,
                  PANEL_LABEL_STRING,
                  "This action will cause unsaved edits to be lost.",
                  0);
```

You may change the label for a message item (as for any type of item) via `PANEL_LABEL_STRING` or `PANEL_LABEL_IMAGE`.

9.5. Buttons

Button items have a label and a menu, but no value.

Button Selection

When the left mouse button is pressed over a button item, the item's rectangle is inverted. When the mouse button is released over a button item, the item's rectangle is painted with a grey background, indicating that the item has been selected and the command is being executed. The grey background is cleared upon return from the notify procedure.

Button Notification

The procedure specified via the attribute `PANEL_NOTIFY_PROC` will be called when the item is selected. The form of the notify procedure for a button is:

```
button_notify_proc(item, event)
    Panel_item item;
    Event      *event
```


Button Image Creation Utility

A routine is provided to create a standardized, button-like image from a string:

```
Pixrect *
panel_button_image(panel, string, width, font)
    Panel    panel;
    char     *string;
    int      width;
    Pixfont  *font;
```

where `width` indicates the width of the button, in character units. The value returned is a pointer to a `pixrect` showing the string with a border drawn around it. If `width` is greater than the length of `string`, the string will be centered in the wider border; otherwise the border will be just wide enough to contain the entire string (i.e., the string will not be clipped). The font is given by `font` — if `NULL`, the font for `panel` is used.

Examples

The first example renders the string in the default system font, found in `/usr/lib/fonts/fixedwidthfonts/screen.r.13`:



```
panel_create_item(panel, PANEL_BUTTON,
    PANEL_NOTIFY_PROC, quit_proc,
    PANEL_LABEL_IMAGE, panel_button_image(panel, "Reset", 0, 0),
    0);
```

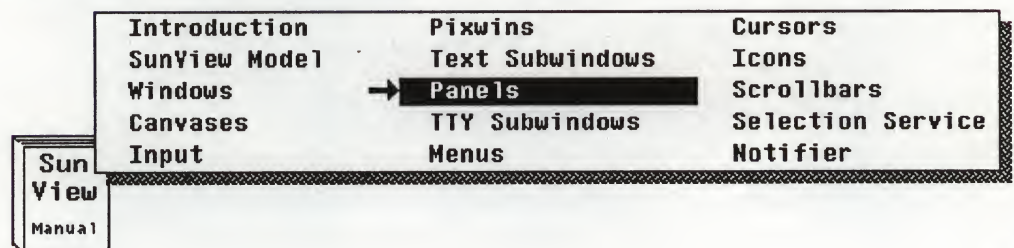
The button below has a bold font and a seven character wide border:



```
bold = pf_open("/usr/lib/fonts/fixedwidthfonts/screen.b.12");
panel_create_item(panel, PANEL_BUTTON,
    PANEL_NOTIFY_PROC, quit_proc,
    PANEL_LABEL_IMAGE, panel_button_image(panel, "Reset", 7, bold),
    0);
```

It is often useful to associate a menu with a button. Figure 9-1 illustrates a button representing an online manual. The menu over the button allows the user to bring up the text for the different chapters:

Figure 9-1 *Associating a Menu With a Button*



To do this, you must write your own event procedure, as described in Section 9.13, *Event Handling*. On receiving a right mouse button down event, display the menu and take the appropriate action depending on which menu item the user selects. For all other events, call the panel's default event procedure.

Here is the code to create the menu and the button, and the event procedure to display the menu:

```
static short    book_array[] = {
#include "book.image"
};
mpr_static(book, 64, 64, 1, book_array);

Menu menu = menu_create( MENU_NCOLS, 3, MENU_STRINGS,
    "Introduction", "Pixwins", "Cursors",
    "SunView Model", "Text Subwindows", "Icons",
    "Windows", "Panels", "Scrollbars",
    "Canvases", "TTY Subwindows", "Selection Service",
    "Input", "Menus", "Notifier", 0,
    0);

panel_create_item(panel, PANEL_BUTTON,
    PANEL_LABEL_IMAGE, &book,
    PANEL_EVENT_PROC, handle_panel_event,
    0);

handle_panel_event(item, event)
    Panel_item item;
    Event *event;
{
    if (event_action(event) ==
        MS_RIGHT && event_is_down(event)) {
        int chapter = menu_show(book_menu, panel, event, 0);
        switch (chapter) {
            case 1: /* Introduction */ break;
            case 2: /* Pixwins */ break;
            ...
            case 15: /* Notifier */ break;
        }
    } else
        panel_default_handle_event(item, event);
}
```


9.6. Choices

Choice items are the most flexible — and complex — item types.⁵⁷ Besides the label, they are composed of:

- a list of either image or string choices (specified via the attributes `PANEL_CHOICE_IMAGES` or `PANEL_CHOICE_STRINGS`).
- a list of *mark-images* — images to be displayed when the corresponding choice is selected (`PANEL_MARK_IMAGES`). The default mark is a push-button with the center inverted.
- a list of *nomark-images* — images to be displayed when the corresponding choice is not selected (`PANEL_NOMARK_IMAGES`). The default nomark image is a non-inverted push-button.

The choices are numbered beginning with zero, and there is no restriction on the number of choices a single choice item may have.

Displaying Choice Items

The attribute `PANEL_DISPLAY_LEVEL` determines which of an item's choices are actually displayed on the screen. The display level may be set to:

- `PANEL_ALL`, (the default) all choices are shown;
- `PANEL_CURRENT`, only the current choice is shown;
- `PANEL_NONE`, no choices are shown. Since the only way of selecting a choice is through the menu, this becomes a label with an associated pop up menu.

If the display level is `PANEL_CURRENT` or `PANEL_ALL`, the choices are placed by default horizontally after the label. You can lay them out vertically below the label by setting `PANEL_LAYOUT` to `PANEL_VERTICAL`. If you want to place the choices or marks more precisely — in order to model a switch or some other special form — you can do so by setting the appropriate attribute, such as `PANEL_CHOICE_XS`, `PANEL_CHOICE_YS`, `PANEL_MARK_XS`, `PANEL_MARK_YS`, etc.

A few words about using the various lists in choice items. The list you give for `PANEL_CHOICE_STRINGS` (or `PANEL_CHOICE_IMAGES`) determines the item's choices.⁵⁸

⁵⁷ For a complete list of the attributes applicable to choice items, see the *Choice Item Attributes* table in are in Chapter 19, *SunView Interface Summary*.

⁵⁸ You must specify at least one choice, so the least you can specify is a single choice consisting of the null string.

The parallel lists `PANEL_CHOICE_FONTS`, `PANEL_MARK_IMAGES`, `PANEL_NOMARK_IMAGES`, `PANEL_MARK_XS`, `PANEL_MARK_YS`, `PANEL_CHOICE_XS`, and `PANEL_CHOICE_YS` are interpreted with respect to the list of choices. For example, the first font given for `PANEL_CHOICE_FONTS` will be used to print the first string given for `PANEL_CHOICE_STRINGS`, the second font will be used for the second string, and so on.

The item below, taken from `iconedit`, shows how parallel lists can be abbreviated:



```
panel_create_item(iced_panel, PANEL_CHOICE,
    PANEL_MARK_IMAGES,      &down_triangle, 0,
    PANEL_NOMARK_IMAGES,    0,
    PANEL_CHOICE_IMAGES,    &square_white, &square_25,
                                &square_root, &square_50,
                                &square_75, &square_black, 0,
    PANEL_VALUE,            2,
    PANEL_CHOICE_XS,        30, 60, 90, 120, 150, 180, 0,
    PANEL_MARK_XS,          34, 64, 94, 124, 154, 184, 0,
    PANEL_CHOICE_YS,        345, 0,
    PANEL_MARK_YS,          363, 0,
    PANEL_NOTIFY_PROC,      proof_background_proc,
    0);
```

The item has six choices, representing the six available background patterns for the proof area. Note, however, that three of the lists, — `PANEL_MARK_IMAGES`, `PANEL_CHOICE_YS` and `PANEL_MARK_YS` all have only one element. When any of the parallel lists are abbreviated in this way, the last element given will be used for the remainder of the choices. So, the 345, 0 in the example above serves as shorthand for 345, 345, 345, 345, 345, 345, 0. All the choice images will appear at y coordinate 345, all the mark images will appear at y coordinate 363, and all the choices will have `down_triangle` as their mark image.

NOTE *You can't specify that a choice or mark-image appear at $x = 0$ or $y = 0$ by using the attributes `PANEL_CHOICE_XS`, `PANEL_CHOICE_YS`, `PANEL_MARK_XS` or `PANEL_MARK_YS`. Since these attributes take null-terminated lists as values, the zero would be interpreted as the terminator for the list. You may achieve the desired effect by setting the positions individually, with the attributes `PANEL_CHOICE_X`, `PANEL_CHOICE_Y`, `PANEL_MARK_X`, or `PANEL_MARK_Y`, which take as values the number of the choice or mark, followed by the desired position.*

Choice Selection

The user can make a selection from a choice item either by selecting the desired choice directly, by selecting from the associated menu, or by selecting the label, which causes the current choice to advance to the next choice (or backup to the previous choice if the shift key is pressed while selecting);

Feedback for choice items comes in two flavors — *inverted*, in which the current choice is shown in reverse video, and *marked*, in which the current choice is indicated by the presence of a distinguishing mark, such as a check-mark or arrow. Specified the type of feedback you want by setting `PANEL_FEEDBACK` to either `PANEL_INVERTED` or `PANEL_MARKED`.

You may also disable feedback entirely, by setting `PANEL_FEEDBACK` to `PANEL_NONE`.

The default feedback is `PANEL_MARKED`, unless the item's display level is current, in which case the feedback is `PANEL_NONE`.

Choice Notification

The procedure specified via the attribute `PANEL_NOTIFY_PROC` will be called when the item is selected. Choice notify procedures are passed the item, the current value of the item, and the event which caused notification:

```
choice_notify_proc(item, value, event)
    Panel_item  item;
    int         value;
    Event       *event;
```

Choice Value

The value passed to the notify procedure is the ordinal number corresponding to the current choice (the choice which the user has just selected). The first choice has ordinal number zero.

Choice Menus

Choice and Toggle items are the only item types for which a menu appears by default. To disable the menu for a particular item, set `PANEL_SHOW_MENU` for that item to `FALSE`.

Choice item menus may be used to represent either a *simple* menu or a *checklist*. The former is a menu of commands, which gives no indication of which command was executed last; the latter is a menu of choices with a check-mark indicating the current choice. Set `PANEL_SHOW_MENU_MARK` to `FALSE` to obtain a simple menu, or `TRUE` to get a checklist.

NOTE The number of menu choices, if set by `PANEL_MENU_CHOICE_STRINGS` or `PANEL_MENU_CHOICE_IMAGES`, must be equal to the number of choices for the item.

Examples

As a basis for our examples we'll take the item in `iconedit` which allows the user to select the drawing mode. The item could have been presented in several different forms.

The simplest call would specify the label and choices as strings, and take the defaults for all other attributes. All the choices will be displayed, and the feedback will be marked, with push-buttons for the mark images:

Drawing Mode: ☐ Points ☐ Line ☐ Rectangle ☐ Circle ☐ Text

```
panel_create_item(panel, PANEL_CHOICE,
    PANEL_LABEL_STRING,    "Drawing Mode:",
    PANEL_CHOICE_STRINGS,  "Points", "Line", "Rectangle",
                           "Circle", "Text", 0,
    0);
```

You can specify a custom mark, such as this small pointer, to indicate the current choice:

Drawing Mode: Points ► Line Rectangle Circle Text

```
static short pointer_array[] = {
#include "pointer.pr"
};
mpr_static(pointer, 16, 16, 1, pointer_array);

panel_create_item(panel, PANEL_CHOICE,
    PANEL_LABEL_STRING,    "Drawing Mode:",
    PANEL_MARK_IMAGES,     &pointer, 0,
    PANEL_NOMARK_IMAGES,   0,
    PANEL_CHOICE_STRINGS,  "Points", "Line", "Rectangle",
                           "Circle", "Text", 0,
    0);
```

Setting `PANEL_FEEDBACK` to `PANEL_INVERTED` produces:

Drawing Mode: Points **Line** Rectangle Circle Text

Often space on the panel is limited, and it is appropriate to save space by only showing the currently selected choice. You can do that by disabling feedback and displaying only the current choice:

Drawing Mode: Line

```
panel_create_item(panel, PANEL_CHOICE,
    PANEL_LABEL_STRING,    "Drawing Mode:",
    PANEL_CHOICE_STRINGS,  "Points", "Line", "Rectangle",
                           "Circle", "Text", 0,
    PANEL_DISPLAY_LEVEL,   PANEL_CURRENT,
    PANEL_FEEDBACK,        PANEL_NONE,
    0);
```

Such an item has the drawback of looking to the user like a text item. One solution to this problem is to provide a distinguishing mark which clearly indicates the item's type, as in:

Drawing Mode: Line

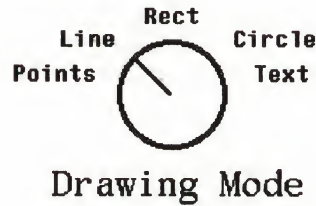
The double-arrow image suggests a cycling motion, indicating to the user that the item is a choice item with more choices available. To get the cycle image, use the special item type `PANEL_CYCLE`:⁵⁹

```
panel_create_item(panel, PANEL_CYCLE,
    PANEL_LABEL_STRING,    "Drawing Mode:",
    PANEL_CHOICE_STRINGS,  "Points", "Line", "Rectangle",
                           "Circle", "Text", 0,
    0);
```

⁵⁹ Note that a cycle item is simply a choice item with some attributes initialized — the display level is set to current and the on-mark is set to the cycle image. Once created, cycle items behave in exactly the same way as choice items.

With some effort, you can use a choice item to model a dial, as in Figure 9-2.

Figure 9-2 *A Dial-Like Choice Item*

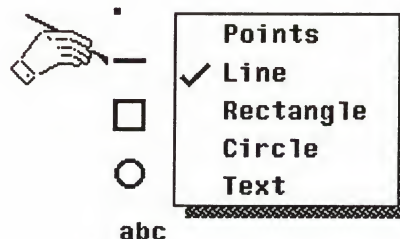


The way to make a such a dial is to make an image for each dial setting, and use these images as the on-marks. Place the on-marks and the choices explicitly — the on-marks in the center, forming the dial, and the choices around the dial's perimeter:

```
panel_create_item(panel, PANEL_CHOICE,
    PANEL_CHOICE_STRINGS, "Points", "Line", "Rect",
                                "Circle", "Text", 0,
    PANEL_MARK_IMAGES,    &dial_1, &dial_2, &dial_3,
                                &dial_4, &dial_5, 0,
    PANEL_NOMARK_IMAGES,  0,
    PANEL_CHOICE_XS,      7, 34, 82, 133, 145, 0,
    PANEL_CHOICE_YS,      53, 33, 20, 33, 53, 0,
    PANEL_MARK_XS,        66, 0,
    PANEL_MARK_YS,        40, 0,
    PANEL_LABEL_STRING,   "Drawing Mode",
    PANEL_LABEL_X,        30,
    PANEL_LABEL_Y,        65,
    PANEL_LABEL_FONT,
    pf_open("/usr/lib/fonts/fixedwidthfonts/gallant.r.19"),
    0);
```

The form which is actually used in shown *iconedit* is Figure 9-3. It employs vertical layout, images for the choices, and strings for the menu:

Figure 9-3 *iconedit's Drawing Mode Choice Item*




```

panel_create_item(panel, PANEL_CHOICE,
    PANEL_LAYOUT,          PANEL_VERTICAL,
    PANEL_CHOICE_IMAGES,   &points, &line, &rectangle,
                           &circle, &text, 0,
    PANEL_MENU_CHOICE_STRINGS, "Points", "Line", "Rectangle",
                           "Circle", "Text", 0,
    PANEL_MARK_IMAGES,     &drawing_hand, 0,
    PANEL_NOMARK_IMAGES,   0,
    0);

```

9.7. Toggles

Toggle items are identical in structure to choice items — they have a label and parallel lists of choices, on-marks and off-marks. They differ from choice items in certain aspects of their display options, their selection behavior and the interpretation of their value. These differences are highlighted below.

Displaying Toggles

Toggle items may have a `PANEL_DISPLAY_LEVEL` of either `PANEL_ALL` — all choices visible, or `PANEL_NONE` — no choices visible. The default is `PANEL_ALL`.

Since there is no notion of the *current* choice for a toggle item, a display level of `PANEL_CURRENT` is not allowed.

Toggle Selection

Toggle items, like choice items, may have either inverted or marked feedback, depending on the value of `PANEL_FEEDBACK`. The default is `PANEL_MARKED`. For inverted feedback, specify `PANEL_INVERTED`. `PANEL_NONE` is not allowed.

Toggle items may be selected by clicking on the desired choice or through the menu. Selecting a choice causes that choice to toggle on or off (change state); other choices are not affected.

If there is only one choice, it may be toggled by selecting the label; if there is more than one choice, selecting the label has no effect.

Toggle Notification

The parameters for the notify procedure are the same as for choice items except that the value passed is a bit mask instead of an integer:

```

toggle_notify_proc(item, value, event)
    Panel_item    item;
    unsigned int  value;
    Event         *event;

```

Toggle Value

The value passed to the notify procedure is a bit mask representing the state of the first 32 choices — if a bit is one, then the corresponding choice is on, if a bit is zero, then the corresponding choice is off. (The least significant bit is bit zero, which maps to choice zero.)

Example

Figure 9-4 illustrates an item which lets you set the *-l*, *-r*, or *-a* flags for the *ls* command:

Figure 9-4 *A Toggle Item*

Format Options:

- ☒ Long
- ☐ Reverse
- ☒ Show all files

```
format_item = panel_create_item(panel, PANEL_TOGGLE,
    PANEL_LABEL_STRING,    "Format Options:",
    PANEL_LAYOUT,          PANEL_VERTICAL,
    PANEL_CHOICE_STRINGS,  "Long",
                           "Reverse",
                           "Show all files",
    0,
    PANEL_TOGGLE_VALUE,    0, TRUE,
    PANEL_TOGGLE_VALUE,    2, TRUE,
    PANEL_NOTIFY_PROC,     format_notify_proc,
    0);
```

You can get or set the value of a particular choice — including choices beyond the first 32 — with `PANEL_TOGGLE_VALUE`. When used to set the value, this attribute takes two values: the index of the choice to set, and the desired value. In the above example, `PANEL_TOGGLE_VALUE` is used to initialize the first and third choices to `TRUE`. To find out the value of the third choice, you would call:

```
value = (int) panel_get(format_item, PANEL_TOGGLE_VALUE, 2);
```

You can also use the attribute `PANEL_VALUE` to set and get the state of a toggle's choices. As mentioned on the previous page, a toggle's value is a bit mask representing the state of the first 32 choices. To facilitate working with the value, you might first define names corresponding to each choice, and a macro to test for the corresponding bit in the value, like this:

```
#define LONG      0
#define REVERSE   1
#define SHOW_ALL  2

#define toggle_bit_on(value, bit)    ((value) & (1 << (bit)))
```

You can then use the value in the notify procedure, as in:


```

format_notify_proc(format_item, value, event)
    Panel_item format_item;
    unsigned int value;
    Event *event;
{
    if (toggle_bit_on(value, LONG)) {
        ...
    }
    if (toggle_bit_on(value, REVERSE)) {
        ...
    }
    if (toggle_bit_on(value, SHOW_ALL)) {
        ...
    }
}

```

Or you can retrieve the value outside of the notify procedure, as in:

```

unsigned value;
value = panel_get_value(format_item);
if (toggle_bit_on(value, LONG)) {
    ...
}

```

Toggle Menus

The menu has as many lines as choices, and each line toggles when selected. In other words, the mark indicating "on" (`PANEL_MENU_MARK_IMAGE`) is alternated with the mark signifying "off" (`PANEL_MENU_NOMARK_IMAGE`) each time the user selects a given line.

To disable the menu, set `PANEL_SHOW_MENU` to `FALSE`.

9.8. Text

Displaying Text Items

The value component of a text item is the string which the user enters and edits. It is drawn on the screen just after the label, as in:

Name: Edward G. Robinson

```

panel_create_item(panel, PANEL_TEXT,
                  PANEL_LABEL_STRING, "Name:",
                  PANEL_VALUE,        "Edward G. Robinson",
                  0);

```

If `PANEL_LAYOUT` is set to `PANEL_VERTICAL`, overriding the default of `PANEL_HORIZONTAL`, the value will be placed below the label.

The number of characters of the text item's value which are displayable on the screen is set via `PANEL_VALUE_DISPLAY_LENGTH`, which defaults to 80 characters. When characters are entered beyond this length, the value string is scrolled one character to the left, so that the most recently entered character is

always visible. As the string scrolls to the left, the leftmost characters move out of the visible display area. The presence of these temporarily hidden characters is indicated by a small left-pointing triangle. So setting the display length to 12 in the above call would produce:

Name: ◀ G. Robinson

As excess characters are deleted, the string is scrolled back to the right, until the actual length becomes equal to the displayed length, and the entire string is visible.

It is sometimes desirable to have a protected field where the user can enter confidential information. The attribute `PANEL_MASK_CHAR` is provided for this purpose. When the user enters a character, the character you have specified as the value of `PANEL_MASK_CHAR` will be displayed in place of the character the user has typed. So setting `PANEL_MASK_CHAR` to “’ * ’” would produce:

Password: *****

If you want to disable character echo entirely, so that the caret does not advance and it is impossible to tell how many characters have been entered, use the space character as the mask. You can remove the mask and display the actual value string at any time by setting the mask to the null character.

The maximum number of characters which can be typed into a text item (independently of how many are displayable) is set via the attribute `PANEL_VALUE_STORED_LENGTH`. Attempting to enter a character beyond this limit causes the field to overflow, and the character is lost. The value string is blinked to indicate to the user that the text item is not accepting any more characters.

The stored length, like the displayed length, defaults to 80 characters.

Text Selection

A panel may have several text items, exactly one of which is *current* at any given time. The current text item is the one to which keyboard input is directed, and is indicated by a caret at the end of the item's value. (If `PANEL_BLINK_CARET` is `TRUE`, the caret will blink as long as the cursor is in the panel.) Selection of a text item (i.e. pressing and releasing the left mouse button anywhere within the item's rectangle) causes that item to become current. A text item also becomes current if it is displayed after being hidden — i.e. if `PANEL_SHOW_ITEM` is set to `TRUE`.

You can find out which text item has the caret, or give the caret to a specified text item, by means of the panel attribute `PANEL_CARET_ITEM`. The call

```
window_set(panel, PANEL_CARET_ITEM, name_item, 0);
```

moves the caret to `name_item`, while


```
(Panel_item)window_get (panel, PANEL_CARET_ITEM);
```

retrieves the item with the caret.

You can rotate the caret through the text items with the following two routines:

```
panel_advance_caret (panel)
    Panel panel;
```

```
panel_backup_caret (panel)
    Panel panel;
```

Advancing past the last text item places the caret at the first text item; backing up past the first text item places the caret at the last text item.

Text Notification

The notification behavior of text items is rather more complex than that of the other item types. You can control whether your notify procedure is called on each input character or only on selected characters. If your notify procedure is called, then the value it returns tells the panel package what to do — whether to insert the character, advance to the next text item, etc.

When your notify procedure will be called is determined by the value of `PANEL_NOTIFY_LEVEL`. Possible values are given in the following table.

Table 9-1 *Text Item Notification*

<i>Notification Level</i>	<i>Causes Notify Procedure to be Called</i>
<code>PANEL_NONE</code>	Never
<code>PANEL_NON_PRINTABLE</code>	On each non-printable input character
<code>PANEL_SPECIFIED</code>	If the input char is found in the string given for the attribute <code>PANEL_NOTIFY_STRING</code>
<code>PANEL_ALL</code>	On each input character

`PANEL_NOTIFY_LEVEL` defaults to `PANEL_SPECIFIED`, and `PANEL_NOTIFY_STRING` defaults to `\n\r\t` (i.e., notification on line-feed, carriage-return and tab).

What happens when the user types a character? The panel package treats some characters specially. `(Meta-C)`,⁶⁰ `(Meta-V)`, and `(Meta-X)` are mapped to the SunView functions `(Copy)`, `(Paste)`, and `(Cut)`, respectively. When the user types these characters, the panel package notices them and performs the appropriate operation, without passing them on to your notify procedure.

The user's editing characters — *erase*, *erase-word* and *kill* — are also treated specially. If you have asked for the character by including it in `PANEL_NOTIFY_STRING`, the panel package will call your notify procedure.

⁶⁰ The *Meta* key is `(Left)` or `(Right)` on the Sun-2 and Sun-3 keyboards. On the type 4 keyboard, the `(Meta)` keys are marked with diamonds `◆`.

After the notify procedure returns, the appropriate editing operation will be applied to the value string. (Note: the editing characters are never appended to the value string, regardless of the return value of the notify procedure.)

Characters other than the special characters described above are treated as follows. If your notify procedure is *not* called, then the character, if it is printable, is appended to the value string. If it is not printable, it is ignored. If your notify procedure *is* called, what happens to the value string, and whether the caret moves to another text item, is determined by the notify procedure's return value. The following table shows the possible return values:

Table 9-2 *Return Values for Text Item Notify Procedures*

<i>Value Returned</i>	<i>Action Caused</i>
PANEL_INSERT	Character is appended to item's value
PANEL_NEXT	Caret moves to next text item
PANEL_PREVIOUS	Caret moves to previous text item
PANEL_NONE	Ignore the input character

If a non-printable character is inserted, it is appended to the value string, but nothing is shown on the screen.

If you don't specify your own notify procedure, the default procedure `panel_text_notify()` will be called at the appropriate time, as determined by the setting of `PANEL_NOTIFY_LEVEL`. The procedure is shown below:

```
Panel_setting
panel_text_notify(item, event)
Panel_item item
Event *event
```

This procedure returns a panel setting enumeration which causes: 1) the caret to move to the next text item on `(RETURN)` or `(TAB)`; 2) the caret to move to the previous text item on `(SHIFT) (RETURN)` or `(SHIFT) (TAB)`; 3) printable characters to be inserted; and 4) all other characters to be discarded.

Writing Your Own Notify Procedure

By writing your own notify procedure, you can tailor the notification behavior of a given text item to support a variety of interface styles. On one extreme, you may want to process each character as the user types it in. For a different application you may not care about the characters as they are typed in, and only want to look at the value string in response to some other button. A typical example is getting the value of a filename field when the user presses the **Load** button.

Text item notify procedures are passed the item and the event which caused notification:

```
Panel_setting
text_notify_proc(item, event)
    Panel_item item;
    Event *event;
```

The input character is referenced by `event_action(event)`.

For example, suppose you want to be notified only when the user types **Esc** or **Control-C** into an item, but you still want them to be able to move to the next item, tab, or select **RETURN**. Create the item as shown below.

```
name_item = panel_create_item(panel, PANEL_TEXT,
    PANEL_LABEL_STRING,    "Enter Name Here:",
    PANEL_NOTIFY_LEVEL,    PANEL_SPECIFIED,
    PANEL_NOTIFY_STRING,   "\n\r\t\033\03",
    PANEL_NOTIFY_PROC,     name_proc,
    0);
```

Note that you must remember to return the appropriate value from your notify procedure. The easiest way to do this is to simply call the default text notify procedure, and return what it returns:

```
Panel_setting
name_proc(item, event)
    Panel_item item;
    Event      *event;
{
    switch (event_action(event)) {
        case ' 33': /* user pressed [Esc] */
            /* special processing of escape */
            return (PANEL_NONE);

        case ' 03': /* user pressed [Ctrl-C] */
            /* special processing of ^C */
            return (PANEL_NONE);

        default:
            return (panel_text_notify(item, event));
    }
}
```

Text Value

As shown in the example under *Displaying Text Items*, you can set the value of a text item at any time via **PANEL_VALUE**. You can also use the **panel_set_value()** macro, as in:

```
panel_set_value(name_item, "Millard Fillmore");
```

The following call retrieves the value of **name_item** into **name**:

```
Panel_item name_item;
char      name[NAME_ITEM_MAX_LENGTH];
...
strcpy(name, (char *)panel_get_value(name_item));
```

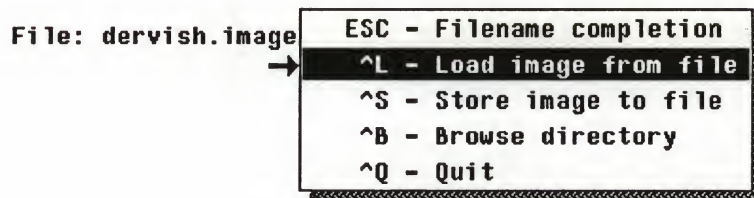
Note that **name_item** should have been created with a **PANEL_VALUE_STORED_LENGTH** not greater than **NAME_ITEM_MAX_LENGTH**, so the buffer **name** will not overflow.

Text Menus

A menu may be associated with a text item by setting `PANEL_SHOW_MENU` to `TRUE`.

Example

One use of text item menus is to make any item-specific “accelerators”, or characters which cause special behavior, visible to the user. This usage of accelerators may be seen in Figure 9-5 which is taken from `iconedit`. The item labelled **File:** holds the name of the file being edited. In addition to typing printable characters, which are appended to the value of the item, the user can type `[Esc]` for filename completion, `[Control-L]` to load an image from the file, `[Control-S]` to store an image to the file, or `[Control-B]` to browse the images in a directory.

Figure 9-5 *A Text Menu*

```
#define ESC 27
#define CTRL_L 12
#define CTRL_S 19
#define CTRL_Q 17
#define CTRL_B 2

filename_item = panel_create_item(panel, PANEL_TEXT,
    PANEL_LABEL_STRING,      "File:",
    PANEL_NOTIFY_LEVEL,      PANEL_ALL,
    PANEL_NOTIFY_PROC,       filename_proc,
    PANEL_VALUE_DISPLAY_LENGTH, 18,
    PANEL_SHOW_MENU,         TRUE,
    PANEL_MENU_CHOICE_STRINGS, "ESC - Filename completion",
                                " ^L - Load image from file",
                                " ^S - Store image to file",
                                " ^B - Browse Directory",
                                " ^Q - Quit",
                                0,
    PANEL_MENU_CHOICE_VALUES,  ESC, CTRL_L, CTRL_S,
                                CTRL_B, CTRL_Q, 0,
    0);
```

The last two attributes specify the menu. `PANEL_MENU_CHOICE_STRINGS` is a null-terminated array of strings to appear as the selectable lines of the menu. The value that the menu returns for each of its lines is specified via `PANEL_MENU_CHOICE_VALUES`. So if the menu line “^L – Load image from file” is selected, the menu will return the value `CTRL_L`. The value returned by the menu is passed directly to the text item, just as if it had been typed at the keyboard.

9.9. Sliders

Displaying Sliders

A slider has four displayable components: the label, the current value, the slider bar, and the minimum and maximum allowable integral values (the range). When `PANEL_SHOW_VALUE` is `TRUE`, the current value is shown in brackets after the label. The font used to render the value is `PANEL_VALUE_FONT`.

The slider bar width in pixels is set with `PANEL_SLIDER_WIDTH`.⁶¹ The minimum and maximum allowable values are set with `PANEL_MIN_VALUE` and `PANEL_MAX_VALUE`. The width of the slider bar corresponding to the current value is filled with grey. The slider bar is always displayed, unless the item is hidden (i.e., `PANEL_SHOW_ITEM` is `FALSE`). When `PANEL_SHOW_RANGE` is `TRUE`, the minimum value of the slider (`PANEL_MIN_VALUE`) is shown to the left of the slider bar and the maximum value (`PANEL_MAX_VALUE`) is shown to the right of the slider bar.

Slider Selection

Only the slider bar of a slider may be selected. When the left mouse button is pressed within the slider bar or the mouse is dragged into the slider bar with the left mouse button pressed, the grey shaded area of the bar will advance or retreat to the position of the cursor. If the mouse is dragged left or right within the slider bar, the grey area will be updated appropriately. If the cursor is dragged outside of the slider bar, the original value of the slider (i.e., the value before the left button was pressed) will be restored.

Slider Notification

Slider notify procedures are passed the item, the item's value at time of notification, and the event which caused notification:

```
slider_notify_proc(item, value, event)
    Panel_item item;
    int         value;
    Event       *event;
```

The notification behavior of a slider is controlled by `PANEL_NOTIFY_LEVEL`. When `PANEL_NOTIFY_LEVEL` is set to `PANEL_DONE`, the notify procedure will be called only when the select button is released within the slider bar. When `PANEL_NOTIFY_LEVEL` is set to `PANEL_ALL`, the notify procedure will be called whenever the value of the slider is changed. This includes:

- when the select button is first pressed within or dragged into the slider bar,
- each time the mouse is dragged within the slider bar,
- when the mouse is dragged outside the slider bar,
- when the select button is released.

⁶¹ If you want to specify the width in characters, use the "column units" macro `ATTR_COLS()` described in Chapter 18, *Attribute Utilities*.

Slider Value

The value of a slider is an integer in the range `PANEL_MIN_VALUE` to `PANEL_MAX_VALUE`. You can retrieve or set a slider's value with the attribute `PANEL_VALUE`.

Example

Figure 9-6 illustrates a typical slider, which might be used to control the brightness of a screen:

Figure 9-6 *A Typical Slider*

Brightness: [75] 0  100

```
panel_create_item(panel, PANEL_SLIDER,
                  PANEL_LABEL_STRING, "Brightness: ",
                  PANEL_VALUE,        75,
                  PANEL_MIN_VALUE,    0,
                  PANEL_MAX_VALUE,    100,
                  PANEL_SLIDER_WIDTH, 300,
                  PANEL_NOTIFY_PROC,  brightness_proc,
                  0);
```

9.10. Painting Panels and Individual Items

To repaint either an individual item or an entire panel, use:

```
panel_paint(panel_object, paint_behavior)
<Panel_item or Panel> panel_object;
Panel_setting      paint_behavior;
```

`paint_behavior` should be either `PANEL_CLEAR`, which causes the rectangle occupied by the panel or item to be cleared prior to repainting, or `PANEL_NO_CLEAR`, which causes repainting to be done without any prior clearing.

You don't have to call `panel_paint()` for items which you create at the same time as you create the panel — when the panel is initially displayed, each of its items will be painted. Note, however, that simply creating a panel item does not cause it to be painted. So items which you create *after* the panel has been initially displayed will not appear until you call `panel_paint()`.

The special attribute `PANEL_PAINT` is provided to allow you to control the "repaint behavior" of an item when one of its attributes is set. `PANEL_PAINT` has three possible values:

- `PANEL_CLEAR` — the item will be automatically cleared and repainted after each call to `panel_set()`.
- `PANEL_NO_CLEAR` — the item will be automatically repainted (without any prior clearing) after each `panel_set()` call.
- `PANEL_NONE` — no automatic repainting will be done.

The default value for `PANEL_PAINT` is `PANEL_CLEAR`. Thus, in the default case, you do not need to call `panel_paint()` after calling `panel_set()`.

You can set the repaint behavior for an item when the item is created, or for all items in the panel when the panel is created. The item's repaint behavior may *not* be reset after the item is created. However, you may temporarily *override* an item's repaint behavior on any call to `panel_set()` by giving a different setting for `PANEL_PAINT`. The examples which follow show two possible repaint policies.

Example 1:

```
item1 = panel_create_item(panel, PANEL_TEXT,
                           PANEL_LABEL_STRING, "Enter Name:",
                           PANEL_VALUE_DISPLAY_LENGTH, 10,
                           PANEL_PAINT,          PANEL_NONE,
                           0);

(begin processing events, etc...)

panel_set(item1, PANEL_ITEM_X, 10, PANEL_ITEM_Y, 50, 0);
panel_set(item1, PANEL_LABEL_IMAGE, &pixrect1, 0);
panel_set(item1, PANEL_VALUE_DISPLAY_LENGTH, 30, 0);
panel_paint(item1, PANEL_CLEAR);
```

Example 2:

```

item2 = panel_create_item(panel, PANEL_TEXT,
    PANEL_LABEL_STRING,      "Enter Name:",
    PANEL_VALUE_DISPLAY_LENGTH, 10,
    0);

(begin processing events, etc...)

panel_set(item2,
    PANEL_ITEM_X, 10,
    PANEL_ITEM_Y, 50,
    PANEL_PAINT, PANEL_NONE,
    0);
panel_set(item2,
    PANEL_LABEL_IMAGE, &pixrect1,
    PANEL_PAINT,      PANEL_NONE,
    0);
panel_set(item2,
    PANEL_VALUE_DISPLAY_LENGTH, 30,
    0);

```

The above two examples each produce the same effect. In the first example, the item's repaint behavior is set to `PANEL_NONE` at creation time, so it is not repainted automatically after the `panel_set()` calls, and no repainting occurs until the call to `panel_paint()`. In the second example, the item's repaint behavior is the default, `PANEL_CLEAR`. This is overridden in the first two `panel_set()` calls, so no repainting occurs. However, it is not overridden in the third call to `panel_set()`, so repainting occurs before that call returns.

As mentioned above, the repaint behavior for all items in a panel can be set when the panel is created, e.g.:

```

window_create(frame, PANEL, PANEL_PAINT, PANEL_NONE, 0);

```

All items created in the above panel will have a repaint behavior of `PANEL_NONE`.

9.11. Iterating Over a Panel's Items

You can iterate over each item in a panel with the two attributes `PANEL_FIRST_ITEM` and `PANEL_NEXT_ITEM`. A pair of macros, `panel_each_item()` and `panel_end_each` are also provided for this purpose. For example, to destroy each item in a panel you would call:

```
Panel_item item;

panel_each_item(browser, item)
    panel_destroy_item(item);
panel_end_each
```

NOTE *Parentheses are not required around the statements to be executed on each iteration. Also, a semicolon is not required after `panel_end_each`.*

9.12. Panel Item Client Data

One attribute applicable to items of all types which should be mentioned is `PANEL_CLIENT_DATA`. You can use this attribute in a variety of ways.

Perhaps the most common use is to associate a unique identifier with each item. This is convenient in the case where you have many items, or where you are creating and destroying items dynamically. If you need to pick one item out of all the items, you can store an identifier (or a class) with it via `PANEL_CLIENT_DATA`, and then query the item directly to find out its identifier or class.

The *dctool* program in Appendix A, *Example Programs*, demonstrates this use of `PANEL_CLIENT_DATA`. The panel buttons for its number keys 0–9 share the same notify procedure. Each button's `PANEL_CLIENT_DATA` holds the ASCII digit displayed on the button; when a button is pushed, the `PANEL_CLIENT_DATA` is retrieved and displayed on the “screen” of the calculator. This saves having a different notify procedure for every button.

You can also use `PANEL_CLIENT_DATA` to associate a pointer to a private structure with an item. For one example of this usage, see the example in the next section under *Writing Your Own Event Handler*. Another application would be to link several items together into a list which is completely under your control.

9.13. Event Handling

This section describes how the panel package handles events.⁶² If you require a behavior not provided by default, you can write your own event handling procedure for either an individual item or the panel as a whole.

Default Event Handling

Using the default event handling mechanism, events are handled for all the panel items in a uniform way. A single routine reads the events, updates an internal state machine, and maps the event to an *action* to be taken by the item. Actions fall into two categories: *previewing* and *accepting*. The previewing action gives the user visual feedback indicating what will happen when he releases the mouse button. The accepting action causes the item's value to be changed and/or its notify procedure to be called, with the event passed as the last argument.

The default event-to-action mapping is given in the following table:

<i>Event</i>	<i>Action</i>
Left button down or drag in w/left button down	Begin previewing
Drag with left button down	Update previewing
Drag out of item rectangle with left button down	Cancel preview
Left button up	Accept
Right button down	Display menu & accept user's selection
Keystroke	Accept keystroke if text item

What actually happens when an item is told to perform one of the above actions depends on the type of the item. For example, when asked to begin previewing, a button item inverts its label, a message item does nothing, a slider item redraws the shaded area of its slider bar, etc.⁶³

Writing Your Own Event Handler

You may want to handle events in a way which is not supported by this default scheme. For example, there is no way to take any action on middle mouse button events. To do so you must extend the event handling functionality by replacing the default event-to-action mapping function for a panel or panel item. Three attributes have been defined for this purpose:

Table 9-3 *Panel Event Handling Attributes*

<i>Attribute</i>	<i>Argument Type</i>	<i>Default Value</i>
PANEL_EVENT_PROC	int (*) ()	panel_default_handle_event ()
PANEL_BACKGROUND_PROC	int (*) ()	panel_default_handle_event ()
PANEL_ACCEPT_KEYSTROKE	boolean	FALSE

An item's PANEL_EVENT_PROC is called when an event falls over the item. The event procedure for an item defaults to that for the panel. Thus you can change the event procedure for all the items in a panel by specifying your own PANEL_EVENT_PROC for the panel before the panel items are created. The arguments passed to the event procedure are the item (or panel) and the event.

⁶² The general SunView input paradigm, including details on the various events, is covered in Chapter 6, Handling Input.

⁶³ For particulars, see the *Selection* subsection under each item type.

The default event procedure, which implements the default event-to-action mapping described on the previous page, is:

```
panel_default_handle_event(object, event)
    <Panel_item or Panel> object;
    Event *event;
```

The panel's `PANEL_BACKGROUND_PROC` is called when an event falls on the background of the panel (i.e. an event whose locator position does not fall over any item). The default panel background procedure is also `panel_default_handle_event()`; however, the various actions are no-ops for the panel. Note that this attribute only applies to a panel; it has no meaning for an individual panel item.

The attribute `PANEL_ACCEPT_KEYSTROKE` determines whether or not an item or panel is interested in keystroke events. If this is `TRUE`, the item or panel under the cursor is given keystroke events as they are generated. The default behavior sends all keystroke events to the text item with the caret, independent of the cursor position.

In addition to the three event related attributes, three event codes have been defined:

- `PANEL_EVENT_DRAG_IN` — the item or panel was entered for the first time with one or more buttons down.
- `PANEL_EVENT_MOVE_IN` — the item or panel was entered for the first time with no mouse buttons down.
- `PANEL_EVENT_CANCEL` — the item or panel is no longer “current” so any operations in progress should be canceled (e.g. cancel previewing).

The panel package will generate these events as appropriate and pass them to the item's event procedure or the panel's background procedure.

The event-to-action mapping is performed by means of a set of *action functions*. If you haven't specified an event procedure for the item, `panel_default_handle_event()` will map events to the appropriate actions by calling one of the action functions. These action functions have been made public so that, if you replace the event procedure for an item, you can ask the panel package to perform one of the default actions by calling the corresponding action function from your new event procedure.

The action functions are given in the table on the following page.

Table 9-4 *Panel Action Functions*

<i>Definition</i>	<i>Description</i>
<code>panel_accept_key(object, event)</code> <Panel or Panel_item> object; Event *event;	Tells a text item to accept a keyboard event. Currently ignored by non-text panel items.
<code>panel_accept_menu(object, event)</code> <Panel or Panel_item> object; Event *event;	Tells an item to display its menu and process the user's selection.
<code>panel_accept_preview(object, event)</code> <Panel or Panel_item> object; Event *event;	Tells an item to do what it is supposed to do when selected, including completing any previewing feedback.
<code>panel_begin_preview(object, event)</code> <Panel or Panel_item> object; Event *event;	Tells an item to begin any feedback which indicates tentative selection.
<code>panel_cancel_preview(object, event)</code> <Panel or Panel_item> object; Event *event;	Tells an item to cancel any previewing feedback.
<code>panel_update_preview(object, event)</code> <Panel or Panel_item> object; Event *event;	Tells an item to update its previewing feedback (e.g. redraw the slider bar for a slider item).

In most of the action routines, only the event's location and shift state are of interest. When previewing, choices, toggles and sliders use the event's location to determine the current value. Choices use the shift state to determine whether to advance or backup the current choice. `panel_accept_key()` is the only action function to make use of the actual event code.

Example

Suppose you are implementing *dbxtool* and want to have the buttons in the command panel execute different commands depending on whether they were selected with the left or middle mouse button. For example, the button labeled **next** might behave as the **step** button if activated with the middle button. When the middle button is depressed, you want to preview an alternate label, and when it is released, you want to execute the dbx command corresponding to the pre-viewed label.

You can get this functionality by replacing the event procedure for each of the button items in the command panel. This could be done either by specifying a default event procedure for all the items when the panel is created:

```
panel = window_create(frame, PANEL,
                      PANEL_EVENT_PROC, dbx_event_proc,
                      0);
```

or by specifying a the event procedure as each panel item is created:

```
panel_create_item(panel, PANEL_BUTTON,
                  PANEL_EVENT_PROC,    dbx_event_proc,
                  0);
```


Whenever one of the buttons gets an event, `dbx_event_proc()` will be called and can then map the events to actions as it sees fit. The code for the new event procedure is given below. Note the use of `PANEL_CLIENT_DATA` to store the images for the two labels for each item.

```
dbx_event_proc(item, event)
    Panel_item item;
    Event      *event;
{
    struct dbx_data *dbx_data; /* data stored with each item */
    Panel          panel;

    /* First get my private data for this item. */
    panel = (Panel) panel_get(item, PANEL_PARENT_PANEL);
    dbx_data = (struct dbx_data *) panel_get(item, PANEL_CLIENT_DATA);

    /* See if this is an event of interest. */
    switch (event_action(event)) {

        /* middle button went up or down */
        case MS_MIDDLE:
            if (event_is_down(event)) {
                /* middle button went down, so change the button's label
                 * image to reflect its middle button action.
                 */
                panel_set(item, PANEL_LABEL_IMAGE, dbx_data->middle_pr, 0);

                /* now begin the normal previewing */
                panel_begin_preview(item, event);
            } else {
                /* middle button went up, so accept the previewing */
                panel_accept_preview(item, event);

                /* now change the image back */
                panel_set(item, PANEL_LABEL_IMAGE, dbx_data->left_pr, 0);
            }
            break;

        /* drag into item with button down */
        case PANEL_EVENT_DRAG_IN:
            if (window_get(panel, WIN_EVENT_STATE, MS_MIDDLE)) {
                /* middle button is down, so treat this as begin preview.
                 */
                panel_set(item, PANEL_LABEL_IMAGE, dbx_data->middle_pr, 0);
                panel_begin_preview(item, event);
            }
            else
                /* we weren't previewing, so
                 * let the default event proc handle it.
                 */
                panel_default_handle_event(item, event);
            break;
    }
}
```

```

/* cancel for some reason */
case PANEL_EVENT_CANCEL:
    if (panel_get(item, PANEL_LABEL_IMAGE) ==
        dbx_data->middle_pr) {
        /* we were previewing -- cancel it.
        */
        panel_cancel_preview(item, event);
        panel_set(item, PANEL_LABEL_IMAGE, dbx_data->left_pr, 0);
    } else
        /* we weren't previewing, so
        * let the default event proc handle it.
        */
        panel_default_handle_event(item, event);
    break;

/* some other event */
default:
    /* we don't care about this event -- let the default
    * event proc handle it.
    */
    panel_default_handle_event(item, event);
}
}

```

The final step is to modify the notify procedure for each button to perform different actions depending on which mouse button was released. The notify procedure for the step/next button, for example, would look like:

```

next_step_notify_proc(item, event)
    Panel_item item;
    Event      *event;
{
    if (event_action(event) == MS_MIDDLE)
        /* do middle button command, "step" */
    else
        /* do left button command, "next" */
}

```

Translating Events from Panel to Window Space

In the case of a scrollable panel, the panel is larger than the subwindow in at least one dimension. If the panel has been scrolled, each point within the subwindow will have one location in the coordinate space of the panel and a different location in the coordinate space of the subwindow. Two functions are provided to translate event coordinates from panel space to window space, and *vice versa*.

If you read your own events with `window_read_event()`,⁶⁴ you must translate the events from window space to panel space with:

⁶⁴ `window_read_event()` is described in Chapter 6, *Handling Input*.


```

Event *
panel_event(panel, event)
    Panel panel;
    Event *event;

```

To go from panel space to window space, use:

```

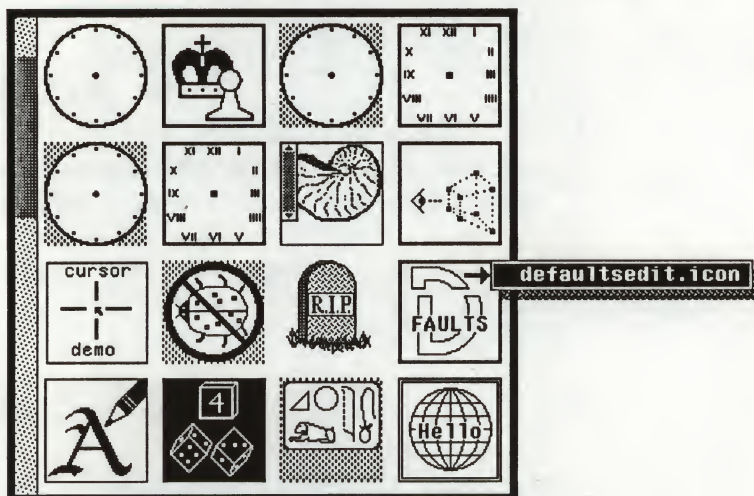
Event *
panel_window_event(panel, event)
    Panel panel;
    Event *event;

```

Example

Figure 9-7 illustrates the image browser from *iconedit*. It serves as an example of when to use `panel_window_event()`. If the user presses the menu button over an image, then he gets a menu showing the name of the file containing the image:

Figure 9-7 *Image Browser Subframe Using panel_window_event()*



In order for the menu to be displayed in the correct place in a panel which has been scrolled, the menu's location must be specified in the coordinates of the subwindow, not of the panel.

The browser is implemented as a panel containing buttons having the images as their labels. The buttons are created each time the user wants to browse a different set of images. When each button is created, the name of the file containing the image is stored as the value of the button's `PANEL_CLIENT_DATA`.

Listed below is the event procedure shared by each button. There is a global menu containing a single menu item, `image_menu_item`. If the event is a right mouse button, the display string for this menu item is set to the file name which was previously stored as the button's `PANEL_CLIENT_DATA`.

Then the event is adjusted from panel space to window space, and the menu is displayed at the proper coordinates. If the user selects from the menu, the button's notify procedure, `browser_items_notify_proc()`, is called, so the effect is the same whether the item is selected through the menu or directly.

```
browser_items_event_proc(item, event)
Panel_item item;
Event *event;
{
    if (event_action(event) == MS_RIGHT) {
        Event *adjusted_event;

        menu_set(image_menu_item,
            MENU_STRING, panel_get(item, PANEL_CLIENT_DATA), 0);

        adjusted_event = panel_window_event(browser, event);

        if (menu_show(image_menu, browser, adjusted_event, 0)) {
            browser_items_notify_proc(item);
            return;
        }
    }
    panel_default_handle_event(item, event);
}
```

Note that for all events other than the right mouse button, the panel's default event procedure is called.

Alerts

Alerts	199
10.1. Introduction to Alerts	199
Uses of Alerts	201
10.2. The Components of an Alert	201
Alert Arrow	201
Multiple-Line Text Message	201
Buttons	201
Positioning	202
Beeping	202
10.3. <code>alert_prompt()</code>	202
10.4. Building an Alert	202
Example 1 — Messages and Simple Buttons	203
Yes and No Buttons	204
Example 2 — Changing Fonts	206
Example 3 — Using Triggers	206



Alerts

This chapter describes the alerts package, which you can use by including the file `<suntool/alert.h>` in your program.

This chapter is divided into three logical sections. Section 1 provides a brief introduction to alerts. Section 2 explains the components that make up alerts. Section 3 gives program fragments that introduce most of the alert attributes.

10.1. Introduction to Alerts

An alert is a pop-up frame that contains a panel to notify a user of problems or changes that require their attention. An alert is easily identified visually by a large black arrow that sweeps into the alert window from the left. A SunView application can use alerts to notify a user that an event has taken place or to verify that a user requested some action. Each alert that pops up has full screen access. That is, the screen is frozen until the user responds to the alert.

Alerts are a replacement for the `menu_prompt()` facility. Some programs will use menu prompts instead of alerts if the user disables alerts in `defaultsedit`. Menu prompts offer a simple box with text, and a maximum of two choices.

Alerts, on the other hand, have a better user interface. Alerts provide an attention-getting alert arrow, buttons, fonts, beeps, a 3-D shadow, and so on. Using alerts, you can offer a user more than two choices of action.

Summary Listing and Tables

To give you a feeling for what you can do with alerts, the following page contains a list of the available alert attributes and functions. Many of these are discussed in the rest of this chapter as they occur in the examples and elsewhere (use the *Index* to check). All are briefly described with their arguments in the alert summary tables in Chapter 19, *SunView Interface Summary*:

- the *Alert Attributes* table begins on page 316;
- the *Alert Functions and Macros* table begins on page 318;

<i>Alert Attributes</i>	
ALERT_BUTTON	ALERT_MESSAGE_STRINGS_ARRAY_PTR
ALERT_BUTTON_FONT	ALERT_NO_BEEPING
ALERT_BUTTON_NO	ALERT_OPTIONAL
ALERT_BUTTON_YES	ALERT_POSITION
ALERT_MESSAGE_FONT	ALERT_POSITION
ALERT_MESSAGE_STRINGS	ALERT_TRIGGER

<i>Alert Functions</i>
<code>alert_prompt(client_frame, event, attributes)</code>

Uses of Alerts

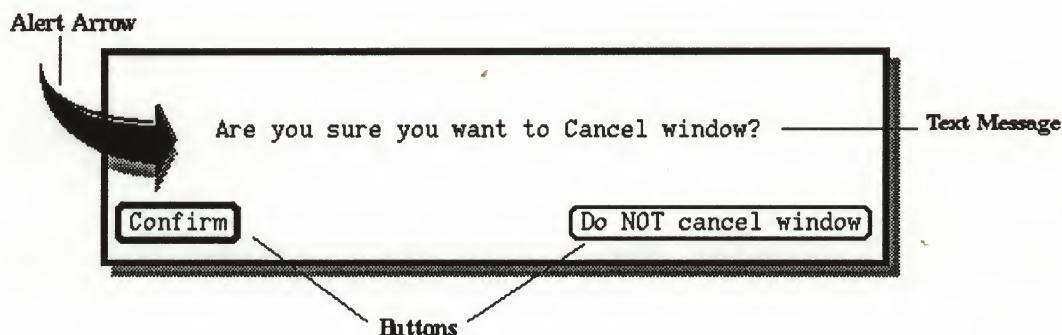
A SunView application uses alerts to display messages to the user, who can then either continue, cancel, or choose a different course of action. Possible uses of alerts include the following:

- Querying whether an action was intended: “Are you sure you want to Quit?”
- Notifying a user of a current state: “Unrecognized file name. No files match specified pattern.”

10.2. The Components of an Alert

Figure 10-1 illustrates the visible components that make up an alert. Each component is described below.

Figure 10-1 *An Alert*



Alert Arrow

Each alert window is identifiable as an alert by the large black arrow that sweeps into the window from the left.

Multiple-Line Text Message

Do you really want to exit SunView?

A multiple-line text message describes why an alert appeared and what to do in order to continue. For example, if the user tries to quit SunView, an alert with the message, “Do you really want to exit SunView?”, will pop up.

Buttons

Cancel

Buttons make it possible to give the user a choice of actions when warning them that an event has taken place. Each button is associated with a string that specifies an action.

Many alerts have a default button which is indicated by a double outline (as in the **Confirm** button above). If an alert has a default button, then the pointer will jump to this button when the alert appears, so that clicking LEFT will take the default action. The pointer is moved back to its original position when the alert goes away. The user can disable *pointer jumping* by setting `SunView/Alert_Jump_Cursor` to disabled in `defaultsedit`.

Positioning

You have three choices for alert placement. The alert may be screen-centered, client-centered, or client-offset.

Beeping

An alert may be specified to pop up with or without a beep. The default is to come up beeping the number of times that is specified in `defaultsed.it`. You may set your alert to come up without a beep even if the user's default *SunView/Alert_Bell* entry in `defaultsed.it` is to come up beeping.

10.3. `alert_prompt()`

There is only one function in the alert package, `alert_prompt()`; it creates an alert, pops it up on the screen, handles user interaction, then takes down the alert and returns a value.

```
int
alert_prompt(client_frame, event, attributes)
    Frame      client_frame;
    Event      *event;
    <attribute-list> attributes;
```

`alert_prompt()` displays an alert whose appearance and behavior is specified by the attribute value list `attributes`. It does not return a value until the user pushes a button in the alert or the default trigger event or its accelerator is seen. By default the alert is positioned over the center of `client_frame`.

If you supply a pointer to an event as `event`, it will be filled in with the user event which dismissed the alert. For example, if the user pushes a button by clicking `LEFT`, `event_action(event)` will be `MS_LEFT`.⁶⁵

The possible status values which `alert_prompt()` returns are:

- `ALERT_YES` — the user pushed the “yes” alert button
- `ALERT_NO` — the user pushed the “no” alert button
- `ALERT_FAILED` — the `alert_prompt()` failed for some reason
- `ALERT_TRIGGERED` — a triggered response occurred
- Some other integer — the user pushed some other button than “yes” or “no.”

10.4. Building an Alert

This section contains code fragments that illustrate most of the attributes for the alerts package. For a complete list and explanation of the alert attributes, see Chapter 19, *SunView Interface Summary*. Each code fragment described below is organized as follows:

- Attributes introduced in the code are described
- An illustration of the alert box is given
- The code is listed and described.

⁶⁵ See Chapter 6, *Handling Input* for an explanation of the Events.

Example 1 — Messages and Simple Buttons

For a complete program example using alerts, see *filer* in Appendix A, *Example Programs*.

This section gives two code fragments in order to illustrate the different button attributes. The buttons allow the user to choose an action. Each alert may contain one or more buttons; the default is for no buttons.

Each button has a name and an associated value. When a user pushes a button, the value associated with the button is returned.

The following attributes are used in the first code fragment. STRINGS ""
ALERT_MESSAGE_STRINGS

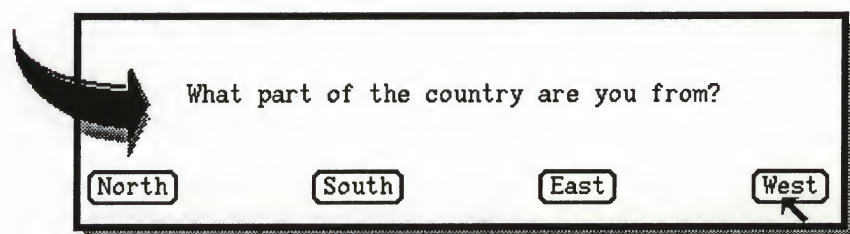
The ALERT_MESSAGE_STRINGS attribute specifies a string or strings to be displayed in the message area of the alert panel.

An example of the syntax for a message is:

```
ALERT_MESSAGE_STRINGS,
    "The text has been edited.",
    "Empty Document will discard these edits. Please confirm",
    0,
```

The ALERT_BUTTON attribute displays a string in a button and associates a value to it. The value specified with the string is returned when the button is pushed. The value may be any integer, but should not be one of the values predefined by the alerts package (ALERT_YES, ALERT_NO, ALERT_FAILED, or ALERT_TRIGGERED). Figure 10-2 illustrates an alert that was built using the attributes ALERT_BUTTON. It contains four buttons and one text string. This example asks the user what part of the country they are from. The program fragment is listed below.

Figure 10-2 A Simple Alert



```
result = alert_prompt(
    (Frame) client_frame,
    (Event*) NULL
    ALERT_MESSAGE_STRINGS
        "What part of the country are you from?",
        0,
    ALERT_BUTTON,    "North",    101,
    ALERT_BUTTON,    "East",     102,
    ALERT_BUTTON,    "West",     103,
    ALERT_BUTTON,    "South",    104,
```



```

    0);

switch (result) {
    case 101:
        /*handle case for someone from the North*/
        break;
    case 102:
        /*handle case for someone from the East*/
        break;
    case 103:
        /*handle case for someone from the West*/
        break;
    case 104:
        /*handle case for someone from the South*/
        break;
    case ALERT_FAILED:
        /*
         * Possibly out of memory or fds;
         * attempt to get information another way
         */
        break;
};

```

Yes and No Buttons

Usually you will want to map your buttons to “yes” and “no” actions. To make this possible, two special buttons are triggered by predefined keyboard accelerators. Yes (confirm, do it) is mapped to the Return key. No (cancel, don't do it) is mapped to the Stop key (usually L1).

The SunView event name for yes is ACTION_DO_IT. The SunView event name for no is ACTION_STOP.

The following attributes are used in this example:

The ALERT_BUTTON_YES attribute associates a string with the accelerated YES button. The value ALERT_YES is returned by alert_prompt() if the user pushes this button, or types Return. Only one instance of this attribute is allowed; subsequent instances are ignored.

The YES button image will have a different button image than the other buttons. It will appear as a regular button image with a double outline.

An example of the syntax is:

```
ALERT_BUTTON_YES, "Confirm, discard edits",
```

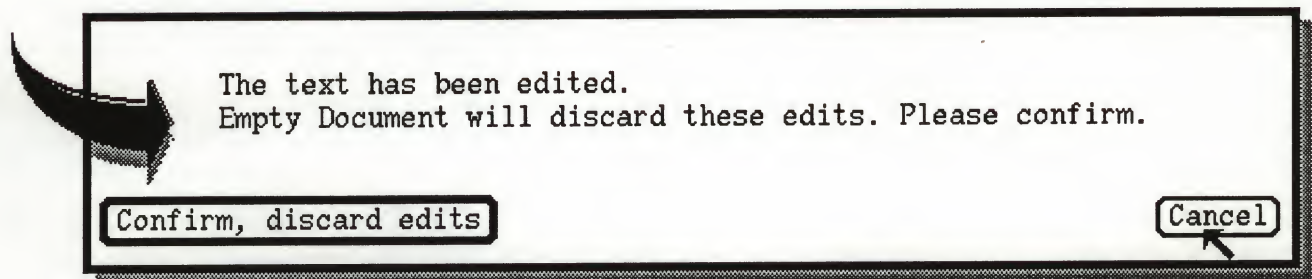
The ALERT_BUTTON_NO attribute associates a string with the accelerated NO button. The value returned if the user pushes this button, or types Stop, will be ALERT_NO. Only one instance of this attribute is allowed; subsequent instances are ignored.

An example of the syntax is:

```
ALERT_BUTTON_NO, "Cancel",
```

Figure 10-3 illustrates the alert that is generated by the following code. It contains two buttons and two text strings. The buttons give the user two choices: to empty a document, discarding any edits they may have made, or to cancel the operation completely.

Figure 10-3 A YES/NO Alert



```
int result;
result = alert_prompt(
    (Frame)window, (Event*)NULL,
    ALERT_MESSAGE_STRINGS,
    "The text has been edited.",
    "Empty Document will discard these edits.\
    Please confirm.",
    0,
    ALERT_BUTTON_YES,      "Confirm, discard edits",
    ALERT_BUTTON_NO,       "Cancel",
    0);
switch(result){
case ALERT_YES:
    /*discard edits*/
    break;
case ALERT_NO:
    /*cancel the Empty Document request */
    break;
case ALERT_FAILED:
    break;
};
```


Example 2 — Changing Fonts

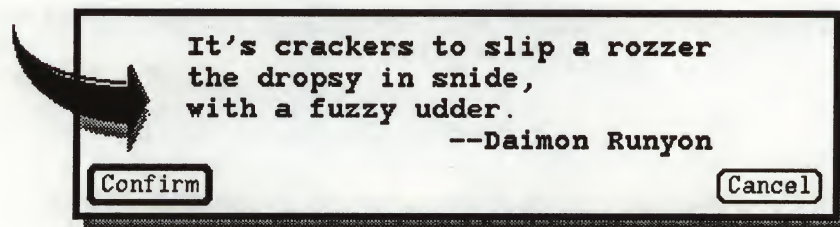
The default font used for alert message text is the Client Frame's font, if one has been specified; or else it is the same as *SunView/Font*. The default font for alert buttons is the same as that specified for menus in *Menus/Font* in *defaultsedit*, or *screen.b.14*, if no default is specified.

You may prefer to use different fonts within alerts. For example, you might want to set off the text in an alert box from the text in the Client's frame by using the **bold** version of the Client Frame's default font.

The `ALERT_MESSAGE_FONT` and `ALERT_BUTTON_FONT` attributes control the font setting for the alert message text and alert buttons, respectively.

Figure 10-4 illustrates an alert in which the message string is printed in `courier.b.16`. The code fragment shown below it illustrates how to set the attribute's value using the font library. It also illustrates the use of multiple message strings.

Figure 10-4 *An Alert with Boldface Message Strings*



```
Event alert_event;
int result = alert_prompt(base_frame, &alert_event,
    ALERT_MESSAGE_STRINGS,
    "It's crackers to slip a rozzer",
    "the dropsy in snide,",
    "with a fuzzy udder.",
    "                                --Daimon Runyon",
    0,
    ALERT_BUTTON_YES, "Confirm",
    ALERT_BUTTON_NO, "Cancel",
    ALERT_MESSAGE_FONT,
    pf_open("/usr/lib/fonts/fixedwidthfonts/cour.b.16"),
    ALERT_POSITION, ALERT_CLIENT_CENTERED,
    0);
```

Example 3 — Using Triggers

Often you will want to give the user the choice of using mouse buttons or keyboard accelerators instead of push buttons to respond to an alert. Triggers give you this option by making it possible to specify an accelerator or mouse action for a choice.

For example, the text window uses an alert to ask the user where to split a window. A left mouse button click is the trigger that responds to this alert.

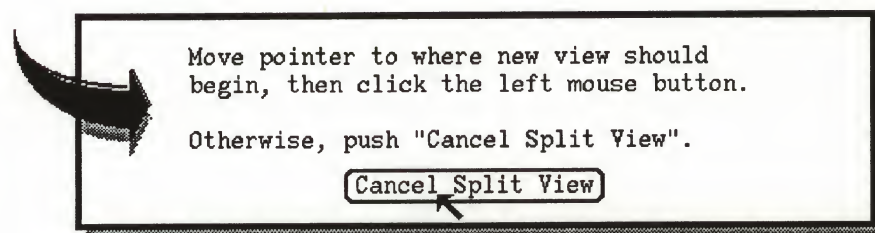
The following attribute is used when specifying a trigger:

The `ALERT_TRIGGER` attribute allows the application to specify a SunView event which should cause the alert to return. The default is not to return a value unless a button has been pushed or the other YES/NO accelerators are seen. When an event is triggered, the value returned will be `ALERT_TRIGGER`. An example of the message syntax is as follows.

```
ALERT_TRIGGER, event,
```

Figure 10-5 illustrates the alert that is generated by the following code. This alert contains one button and a triggered response. When this alert comes up, the user may split the existing window into two windows, or can dismiss the alert by pushing the **Cancel New Window** button. This example also shows how alerts can effectively use an event to collect information about the way a user reacted to an alert. See Chapter 6, *Handling Input*, for a full explanation and list of all possible events.

Figure 10-5 *An Alert Using Triggers and Events*




```

Event event;
int  result;

result = alert_prompt(
    (Frame)window,
    &event,
    ALERT_NO_BEEPING,          1,
    ALERT_MESSAGE_STRINGS,
    "Move pointer to where new window should",
    "appear, then click the left mouse button.",
    "Otherwise, push \"Cancel New Window.\"\\",
    0,
    ALERT_BUTTON_NO,          "Cancel New Window",
    ALERT_TRIGGER,            MS_LEFT,
    0);
switch (result) {
case ALERT_TRIGGERED:
    (void) create_new_window_at_pos(event_x(&event),
                                    event_y(&event)),
    break;
case ALERT_NO:
    break; /* don't create new window */
case ALERT_FAILED:
    /* alert failed, possibly out of memory or fds */
}

```

You may specify in your code to have an alert pop up without a beep as shown above. Generally, beeping is reserved for any event which occurs unexpectedly. If the alert is in response to a user request, it should not beep.

The following attribute is used to specify no beeping for an alert.

The `ALERT_NO_BEEPING` attribute allows the SunView application to specify that no beeping should take place regardless of `defaultsedit` setting. The default for this option is `FALSE`; that is, beep as many times as the defaults database specifies.

TTY Subwindows

TTY Subwindows	211
11.1. Creating a TTY Subwindow	213
11.2. Driving a TTY Subwindow	213
ttysw_input ()	213
ttysw_input ()	214
Example: <i>tty_io</i>	214
11.3. TTY Subwindow Escape Sequences	214
Standard ANSI Escape Sequences	214
Special Escape Sequences	215
Example: <i>tty_io</i>	215
11.4. Reading and Writing to a TTY Subwindow	215
11.5. The Program in the TTY Subwindow	215
TTY_PID	215
Talking Directly to the TTY Subwindow	216
An Example	216



TTY Subwindows

The `tty` (or *terminal emulator*) subwindow emulates a standard Sun terminal, the principal difference being that the row and column dimensions of a `tty` subwindow can vary. You can run arbitrary programs in a `tty` subwindow; perhaps its main use is to run a shell within a window.

To see `tty` subwindows in use, run the standard tools `shelltool(1)` and `gfxtool(1)`.

Header Files

Programs using `tty` subwindows must include the file `<suntool/tty.h>`.

Summary Listing and Tables

To give you a feeling for what you can do with `tty` subwindows, the following page contains lists of the available `tty` subwindow attributes, functions and macros. Many of these are discussed in the rest of this chapter and elsewhere (use the *Index* to check). All are briefly described with their arguments in the `tty` subwindow summary tables in Chapter 19, *SunView Interface Summary*:

- the *TTY Subwindow Attributes* table begins on page 376;
- the *TTY Subwindow Functions* table begins on page 376;
- the *TTY Subwindow Special Escape Sequences* table begins on page 377.

TTY Subwindow Attributes

TTY_ARGV	TTY_PAGE_MODE
TTY_CONSOLE	TTY_QUIT_ON_CHILD_DEATH

TTY Subwindow Functions

ttysw_input(tty, buf, len)	ttysw_output(tty, buf, len)
----------------------------	-----------------------------

11.1. Creating a TTY Subwindow

Like all SunView windows, you create a tty subwindow by calling `window_create()` with the appropriate type parameter, as in:

```
Tty tty;
tty = window_create(frame, TTY, 0);
```

By default, the tty subwindow will fork a shell. If you want to start the tty subwindow with another program, say *vi*, you can do so by specifying the name of the program to run via the `TTY_ARGV` attribute:

```
#include <suntool/sunview.h>
#include <suntool/tty.h>

char *my_argv[] = { "vi", 0 };

main()
{
    Tty tty;
    Frame frame;

    frame = window_create(0, FRAME, 0);
    tty = window_create(frame, TTY, TTY_ARGV, my_argv, 0);
    window_main_loop(frame);
}
```

NOTE *You can only have one tty subwindow per process.*

11.2. Driving a TTY Subwindow

You can drive the terminal emulator programmatically. There are procedures both to send input to the terminal emulator (as if the user had typed it in the tty subwindow) and to send output (as if a program running in the tty subwindow had output it). The two effects are similar to the `mapi / mapo` functions in `~/ .ttyswrc` that permit a user to bind a character sequence to a function key.⁶⁶

`ttysw_input()`

You can send input to a tty subwindow programmatically with the function:

```
int
ttysw_input(tty, buf, len)
    Tty    tty;
    char *buf;
    int    len;
```

`ttysw_input()` appends the character sequence in `buf` that is `len` characters long onto `tty`'s input queue. It returns the number of characters accepted. The characters are treated as if they were typed from the keyboard. `ttysw_input()` provides a simple way for a window program to send input to a program running in its tty subwindow.

⁶⁶ See `shelltool(1)` in the *SunOS Reference Manual*.

`ttysw_input()`

Use `ttysw_output()` to output to a tty subwindow.

```
int
ttysw_output(tty, buf, len)
    Tty    tty;
    char *buf;
    int    len;
```

`ttysw_output()` runs the character sequence in `buf` that is `len` characters long through the terminal emulator of `tty`. It returns the number of characters accepted. The effect is similar to executing

```
echo character_sequence > /dev/ttyN
```

where `ttyN` is the pseudo-tty associated with the tty subwindow. One use of `ttysw_output()` is to send the escape sequences listed in the next section to the tty subwindow.

Example: `tty_io`

Appendix A, *Example Programs*, gives the listing for `tty_io`, a program which uses `tty_output()` to output strings of characters to a tty subwindow.

11.3. TTY Subwindow Escape Sequences

Standard ANSI Escape Sequences

The tty subwindow accepts the same ANSI escape sequences as the raw Sun console,⁶⁷ with the following few exceptions:

- The effect of the bell control character CTRL-G (0x07) in a tty subwindow depends on how the user has set the two options *Audible_Bell* and *Visible_Bell* in the *SunView* category in `defaultsedit(1)`. If *Audible_Bell* is *Enabled*, the bell will ring. If *Visible_Bell* is *Enabled*, the window will flash.
- The graphics rendition sequences ESC [4m (underline) and ESC [1m (bold "extra-bright") operate correctly. On the Sun console, these sequence always invert subsequent characters, whereas the tty subwindow only inverts when sent ESC [7m (stand-out).
- The effect of the bold "extra-bright" graphics rendition sequence ESC [1m in a tty subwindow depends on the user's setting for the *Bold_style* option in the *Tty* category of `defaultsedit`.
- Unsupported graphics rendition mode escape sequences have the same effect as that chosen for bold "extra-bright". On the Sun console, everything inverts.
- The Set Scrolling sequence ESC [0r, which enables vertical wrap mode on the Sun terminal, has no effect in a tty subwindow.

⁶⁷ See the `console(4s)` manual page in the *SunOS Reference Manual* for a full list of escape sequences.

- You can modify `termcap(5)` if you need further control over what gets displayed in the different modes. The two-character `termcap` symbols for each of the modes are:

```
so    standout
us    underline
md    bold (extra bright)
```

Special Escape Sequences

Escape sequences have been defined by which the user can get and set attributes of both the tty subwindow and the frame which contains it. For example, the user can type an escape sequence to open, close, move or resize the frame, change the label of the frame or the frame's icon, etc. These escape sequences are described in Table 19-33, *TTY Subwindow Special Escape Sequences*, in Chapter 19, *SunView Interface Summary*.

Example: `tty_io`

For an example of setting the frame's label via a tty subwindow escape sequence, see the program `tty_io`, listed in Appendix A, *Example Programs*.

11.4. Reading and Writing to a TTY Subwindow

You cannot use the tty subwindow's file descriptor returned by `WIN_FD` to read and write characters to it. You can use `TTY_TTY_FD` attribute to get the file descriptor of the pseudo-tty associated with the tty subwindow. You can then use this to read and write to the pseudo-tty using standard UNIX I/O routines. Note that `TTY_TTY_FD` is the file descriptor of the pseudo-tty, not the file descriptor of the tty subwindow returned by `WIN_FD`. The latter is used for low-level window manipulation procedures.

11.5. The Program in the TTY Subwindow

You use the `TTY_ARGV` attribute to pass the name of the program to run to the tty subwindow. The program runs as a forked child in the tty subwindow.

`TTY_PID`

You can use `TTY_PID` to monitor the state of the child process running in the tty window via the Notifier using `L_notify_interpose_wait3_func()`. The client's `wait3()` function gets called when the state of the process in the tty subwindow changes. The setup is something like this:

```
#include <sys/wait.h>
static Notify_value my_wait3();

...
    ttysw = window_create(base_frame,      TTY,
                          TTY_QUIT_ON_CHILD_DEATH, FALSE,
                          TTY_ARGV,        my_argv,
                          0);
    child_pid = (int>window_get(ttysw, TTY_PID);
    notify_interpose_wait3_func(ttysw, my_wait3, child_pid);
...
```

The `wait3()` function can then do something useful, such as destroying the tty window or starting up another process in the tty subwindow. Here is a code fragment that detects the death of its tty subwindow's child. It turns off the default behavior of a tty subwindow, which is to quit when the child process dies.


```

static Notify_value
my_wait3(ttysw, pid, status, rusage)
    Tty          ttysw;
    int          pid;
    union wait    *status;
    struct rusage *rusage;
{
    int    child_pid;

    notify_next_wait3_func(ttysw, pid, status, rusage);
    if (! (WIFSTOPPED(*status))) {
        window_set(ttysw,
                    TTY_QUIT_ON_CHILD_DEATH,    FALSE,
                    TTY_ARGV,                    my_argv,
                    0);
        child_pid = (int)window_get(ttysw, TTY_PID);
        notify_interpose_wait3_func(ttysw, my_wait3, child_pid);
    }
    return NOTIFY_DONE;
}

```

You can set TTY_PID as well as get it, but if you set it then you are responsible for setting the `notify_interpose_wait3_func()` to catch the child's death, and for making the standard input and standard output of the child go to the pseudo-tty.

Talking Directly to the TTY Subwindow

If you set TTY_ARGV to TTY_ARGV_DO_NOT_FORK, this tells the system not to fork a child in the tty subwindow. In combination with TTY_FD, this allows the tool to use standard I/O routines to read and write to the tty subwindow.⁶⁸ This simplifies porting terminal-oriented graphics programs, which interact with the user on the model of *write a prompt... read a reply*, to SunView. However, in most cases you should redesign programs to use a real windowing interface made up of SunView components.

An Example

The *typein* program in Appendix A, *Example Programs* reads and writes directly to its tty subwindow, using SunView's `window_main_loop()` control structure.

The following example preserves the flow of control of a typical UNIX application, using `notify_do_dispatch()` to ensure that the Notifier gets called. Read Section 17.6, *Porting Programs to SunView*, for more information on using the Notifier in this way.

```

#define BUFSIZE 1000
static int      my_done;
static Notify_value
my_notice_destroy(frame, status)
    Frame        frame;
    Destroy_status status;

```

⁶⁸ This capability makes obsolete the work-around required in the 3.0 and 3.2 releases of SunView if you

```

{
    if (status != DESTROY_CHECKING) {
        my_done = 1;
        (void)notify_stop();
    }
    return (notify_next_destroy_func(frame, status));
}

main(argc, argv)
    int     argc;
    char    *argv[];
{
    Frame    base_frame;
    Tty      ttysw;
    int      tty_fd;
    char     buf[BUFSIZE];

    my_done = 0;
    base_frame = window_create(NULL, FRAME,
        FRAME_ARGC_PTR_ARGV,    &argc, argv,
        0);

    ttysw = window_create(base_frame,    TTYSW,
        TTY_ARGV,                TTY_ARGV_DO_NOT_FORK,
        0);
    tty_fd = (int>window_get(ttysw, TTY_TTY_FD);
    dup2(tty_fd, 0);
    dup2(tty_fd, 1);

    (void)notify_interpose_destroy_func(base_frame, my_notice_destroy);
    window_set(base_frame, WIN_SHOW, TRUE, 0);
    (void)notify_do_dispatch();

    puts(prompt_to_user);
    while (gets(buf)) {
        if (my_done) /* continue until destroyed */
            break;
        /*
         * This is where the meat of the program
         * would be if this were a real program.
         */
        puts(buf);
    }
    exit(0);
}

```

wanted a window program to read and write from its own tty subwindow.

Menus

Menus	221
12.1. Basic Menu Usage	224
12.2. Components of Menus & Menu Items	228
Menus	228
Visual Components	228
Generate Procedures	228
Notify Procedures	228
Client Data	228
Menu Items	228
Menu Items	228
Representation on the Screen	228
Item Values	229
Item Generate Procedures	229
Item Action Procedures	229
Client Data	229
Item Margins	230
12.3. Examples	230
12.4. Item Creation Attributes	237
12.5. Destroying Menus	238
12.6. Searching for a Menu Item	239
12.7. Callback Procedures	240
Flow of Control in menu_show()	240
Generate Procedures	242



Menu Item Generate Procedure	243
Menu Generate Procedure	244
Pull-right Generate Procedure	246
Notify/Action Procedures	247
12.8. Interaction with Previously Defined SunView Menus	248
Using an Existing Menu as a Pull-right	248
12.9. Initial and Default Selections	249
12.10. User Customizable Attributes	250

Menus

The SunView menu package allows you to chain individual menus together into a collection known as a *walking menu*. A menu contains *menu items*, some of which may have a small arrow pointing to the right. This indicates to the user that if he or she slides the mouse to the right of that item, a *pull-right* menu will appear. Menus can be strung together in this fashion, so that the user “walks” to the right down the chain of menus in order to make a selection.

The definitions necessary to use walking menus are found in the file `<suntool/walkmenu.h>`, which is included by default when you include the file `<suntool/sunview.h>`.

The most useful sections to read first are the first three. Section 12.1, *Basic Menu Usage*, introduces the basic routines and gives some simple examples. Section 12.2, *Components of Menus & Menu Items*, outlines the components of menus and menu items and introduces common terms. Section 12.3, *Examples*, gives more examples of using menus. Section 12.7, *Callback Procedures*, is for advanced users who need to understand the subtleties of the callback mechanism.

The listing for *font_menu*, a program which builds on some of the examples given throughout the chapter, is given in Appendix A, *Example Programs*.

Summary Listing and Tables

To give you a feeling for what you can do with menus, the following two pages list the available menu attributes, functions and macros. Many of these are discussed in the rest of this chapter and elsewhere (use the *Index* to check). All are briefly described with their arguments in the menu summary tables in Chapter 19, *SunView Interface Summary*:

- the *Menu Attributes* table begins on page 335;
- the *Menu Item Attributes* table begins on page 339;
- the *Menu Functions* table begins on page 341.

Menu Attributes

MENU_ACTION_IMAGE	MENU_LAST_EVENT
MENU_ACTION_ITEM	MENU_LEFT_MARGIN
MENU_APPEND_ITEM	MENU_MARGIN
MENU_BOXED	MENU_NCOLS
MENU_CENTER	MENU_NITEMS
MENU_CLIENT_DATA	MENU_NROWS
MENU_COLUMN_MAJOR	MENU_NOTIFY_PROC
MENU_CLIENT_DATA	MENU_NTH_ITEM
MENU_DESCEND_FIRST	MENU_PARENT
MENU_DEFAULT	MENU_PULLRIGHT_DELTA
MENU_DEFAULT_ITEM	MENU_PULLRIGHT_IMAGE
MENU_DEFAULT_SELECTION	MENU_PULLRIGHT_ITEM
MENU_FIRST_EVENT	MENU_REMOVE
MENU_FONT	MENU_REMOVE_ITEM
MENU_GEN_PROC	MENU_REPLACE
MENU_GEN_PULLRIGHT_IMAGE	MENU_REPLACE_ITEM
MENU_GEN_PULLRIGHT_ITEM	MENU_RIGHT_MARGIN
MENU_IMAGE_ITEM	MENU_SELECTED
MENU_IMAGES	MENU_SELECTED_ITEM
MENU_INITIAL_SELECTION	MENU_SHADOW
MENU_INITIAL_SELECTION_EXPANDED	MENU_STAY_UP
MENU_INITIAL_SELECTION_SELECTED	MENU_STRINGS
MENU_INSERT	MENU_STRING_ITEM
MENU_INSERT_ITEM	MENU_TITLE_IMAGE
MENU_ITEM	MENU_TITLE_ITEM
MENU_JUMP_AFTER_NO_SELECTION	MENU_TYPE
MENU_JUMP_AFTER_SELECTION	MENU_VALID_RESULT

Menu Item Attributes

MENU_ACTION_IMAGE†	MENU_INACTIVE
MENU_ACTION_ITEM†	MENU_INVERT
MENU_ACTION_PROC	MENU_LEFT_MARGIN†
MENU_APPEND_ITEM†	MENU_MARGIN†
MENU_BOXED†	MENU_PARENT†
MENU_CENTER†	MENU_PULLRIGHT
MENU_CLIENT_DATA†	MENU_PULLRIGHT_IMAGE†
MENU_FEEDBACK	MENU_PULLRIGHT_ITEM†
MENU_FONT†	MENU_RELEASE
MENU_GEN_PROC†	MENU_RELEASE_IMAGE
MENU_GEN_PROC_IMAGE	MENU_RIGHT_MARGIN†
MENU_GEN_PROC_ITEM	MENU_SELECTED†
MENU_GEN_PULLRIGHT	MENU_STRING†
MENU_GEN_PULLRIGHT_IMAGE†	MENU_STRING_ITEM†
MENU_GEN_PULLRIGHT_ITEM†	MENU_TYPE†
MENU_IMAGE	MENU_VALUE
MENU_IMAGE_ITEM†	

Menu Functions

```
menu_create(attributes)
menu_create_item(attributes)
menu_destroy(menu_object)
menu_destroy_with_proc(menu_object, destroy_proc)
    void (*destroy_proc)();
menu_find(menu, attributes)
menu_set(menu_object, attributes)
menu_show(menu, window, event, 0)
menu_return_item(menu, menu_item)
menu_return_value(menu, menu_item)
```


12.1. Basic Menu Usage

The basic usage of menus is to first create the menu with `menu_create()`, then display it when desired with `menu_show()`:

```
Menu
menu_create(attributes)
    <attribute-list> attributes;

caddr_t
menu_show(menu, window, event, 0)
    Menu      menu;
    Window    window;
    Event      *event;
```

Like the creation routines for other SunView objects, `menu_create()` takes a null-terminated attribute list and returns an opaque handle. `menu_show()` displays the menu, gets a selection from the user, and, by default, returns the *value* of the menu item the user has selected. `window` is the handle of the window over which the menu is displayed; `event`⁶⁹ is the event which causes the menu to come up. The final argument is provided so that attributes may be passed in the future; at present it is ignored.

Use the routines `menu_set()` and `menu_get()` to modify and retrieve the values of attributes for both menus and menu items:

```
int
menu_set(menu_object, attributes)
    <Menu or Menu_item> menu_object;
    <attribute-list> attributes;

caddr_t
menu_get(menu_object, attribute[, optional_arg])
    <Menu or Menu_item> menu_object;
    Menu_attribute attribute;
    caddr_t optional_arg;
```

All the attributes applying to menus and menu items are listed in the two corresponding tables *Menu Attributes* and *Menu Item Attributes* in Chapter 19, *SunView Interface Summary*. Common attributes applying to both menus and menu items appear in both tables.

The pages which follow contain three examples of basic menu usage.

⁶⁹ Canvases and panels have their own coordinate spaces separate from the window's coordinate space. Note that `event` is in the coordinate space of the window, not of the canvas or panel.

Example 1:



Let's take a very simple example — a menu with two selectable items represented by the strings 'On' and 'Off':

```
on_off_menu = menu_create(MENU_STRINGS, "On", "Off", 0,
                          0);
```

The attribute `MENU_STRINGS` takes a list of strings and creates an item for each string. Note that the first zero in the above call terminates the list of strings, and the second zero terminates the entire attribute list.

CAUTION

The menu package, in contrast to the panel package, does not save strings which you pass in. So you should either pass in the address of a constant, as in the example above, or static storage, or storage which you have dynamically allocated.

Typically you call `menu_show()` from an event procedure,⁷⁰ upon receiving the event which is to cause display of the menu. In the code fragment below, we display the menu on right button down:

```
...
case MS_RIGHT:
    menu_show(on_off_menu, window, event, 0);
    break;
...
```

`menu_show()`, by default, returns the value of the item which was selected. If the item was created with `MENU_STRINGS` its value defaults to its ordinal position in the menu, starting with 1.⁷¹ So in the above example, selecting 'On' would cause 1 to be returned, while selecting 'Off' would cause 2 to be returned.

You can specify that `menu_show()` return the item itself, rather than return the value of the selected item. Do this by setting `MENU_NOTIFY_PROC` to the predefined notify procedure⁷² `menu_return_item()`, as in:

```
menu_set(on_off_menu,
        MENU_NOTIFY_PROC, menu_return_item,
        0);
```

⁷⁰ See Chapter 6, *Handling Input*, for a discussion of event procedures.

⁷¹ The value of menu items not created with `MENU_STRINGS` defaults to zero. You can explicitly specify the values for menu items via the attributes `MENU_IMAGE_ITEM`, `MENU_STRING_ITEM`, or `MENU_VALUE`.

⁷² Notify procedures are covered in detail in Section 12.7, *Callback Procedures*.

Example 2:

It's easy to build up more complex menus out of simple ones. The next example creates a menu with two items, 'Bold' and 'Italic', each of which shares the on-off menu from the previous example as a pull-right:

```
menu = menu_create(MENU_ITEM,
                   MENU_STRING,  "Bold",
                   MENU_PULLRIGHT, on_off_menu,
                   0,
                   MENU_ITEM,
                   MENU_STRING,  "Italic",
                   MENU_PULLRIGHT, on_off_menu,
                   0,
                   0),
```

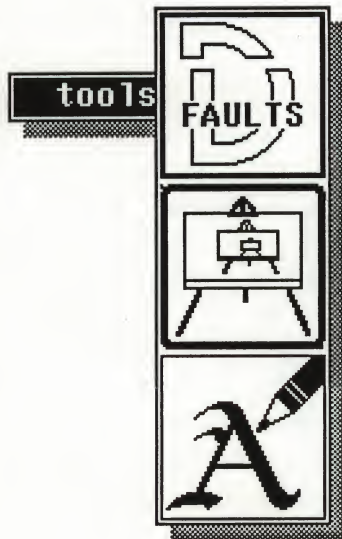
The most flexible way to create a menu item in-line in a `menu_create()` call is by using `MENU_ITEM`. In contrast to `MENU_STRING`, which allows you to specify only the display strings of the items, `MENU_ITEM` takes as its value a null-terminated attribute list which may contain any attributes applying to menu items.⁷³

The value of `MENU_STRING` is the item's display string; the value of `MENU_PULLRIGHT` is the handle of the item's pull-right menu. (Note that you must already have created the menu before giving it as the value for `MENU_PULLRIGHT`.)

Example 3:

The menu package can accommodate images as well as strings. The example below creates a menu with a single item labelled 'tools'. When the user pulls right, he brings up a menu showing the icons of three SunView tools — `defaultsedit`, `iconedit`, and `fontedit`.

⁷³ For a complete list of such attributes, see the *Menu Item Attributes* table in Chapter 19, *SunView Interface Summary*.



In order to pass an image into the menu package you need a pointer to a *memory pixrect* containing the image. One common way to create such an image is by first using *iconedit* to create the image and save it to a file. You then include the file in your program, and use the `mpr_static()` macro to create a memory *pixrect*:

```
static short d_defaults[] = {
#include <images/defaultsedited.icon>
};
mpr_static(defaults_pr, 64, 64, 1, d_defaults);

static short d_icon[] = {
#include <images/iconedit.icon>
};
mpr_static(icon_pr, 64, 64, 1, d_icon);

static short d_font[] = {
#include <images/fontedit.icon>
};
mpr_static(font_pr, 64, 64, 1, d_font);

tool_menu = menu_create(MENU_IMAGES,
                        &defaults_pr, &icon_pr, &font_pr, 0,
                        0);
menu = menu_create(MENU_ITEM,
                  MENU_STRING, "tools",
                  MENU_PULLRIGHT, tool_menu,
                  0,
                  0);
```

The attribute `MENU_IMAGES` is analogous to `MENU_STRINGS`. It takes a list of images (pointers to *pixrects*) and creates a menu item for each image.

12.2. Components of Menus & Menu Items

This section gives an overview of the most important components of menus and menu items. Detailed discussion and examples follow later in the chapter.

Menus

Visual Components

The text for a menu is rendered in the menu's font, which you may specify via `MENU_FONT`. A menu has a *shadow*; you can specify the shadow's pattern, or disable the shadow entirely, via `MENU_SHADOW`. You can give a title to a menu via `MENU_TITLE_IMAGE` or `MENU_TITLE_ITEM`.⁷⁴ By default, a menu's items are laid out vertically; you can specify that the items be laid out horizontally or in a two-dimensional matrix via `MENU_NCOLS` and `MENU_NROWS`.

Generate Procedures

You may specify a *generate procedure* for a menu, which will be called just before the menu is displayed. This allows you to implement context-sensitive menus by dynamically modifying the menu, or even replacing it entirely.⁷⁵

Notify Procedures

The menu's *notify procedure* is called after the user makes a selection. By using a notify procedure, you can perform an action or alter the result or alter the result to be returned by `menu_show()`.⁷⁶

Client Data

The menu's *client data* field, accessible through `MENU_CLIENT_DATA`, is reserved for the application's use. You can use this attribute to associate a unique identifier, or a pointer to a private structure, with a menu.

Menu Items

A menu contains an array of items. To retrieve a menu's *n*th item, use `MENU_NTH_ITEM`. To retrieve the total number of items in a menu use `MENU_NITEMS`.

The same menu item can appear in more than one menu.

CAUTION Menu items, unlike panel items, are counted starting with one.

Menu Items

Representation on the Screen

A menu item is either displayed as a string or an image (a pointer to a `pixrect`). If the item has another menu associated with it using the `MENU_PULLRIGHT` attribute, then it is a pull-right item.

⁷⁴ The title is nothing more than an inverted, non-selectable item. It does not automatically appear at the top of the menu — it is your responsibility to position it where you want it.

⁷⁵ See example 8 in Section 12.7, *Callback Procedures*, later in the chapter.

⁷⁶ Notify procedures are discussed in detail in Section 12.7, *Callback Procedures*.

Item Values

Each menu item has a *value*. By default an item's value is the initial ordinal position of the item if it was created with `MENU_STRINGS`; otherwise the default value is zero. You can set an item's value explicitly when you create the item with `MENU_STRING_ITEM` or `MENU_IMAGE_ITEM`. You can also explicitly set an item's value with `MENU_VALUE`. However, if an item is a pull-right, then its `MENU_VALUE` is the value of its pull-right menu. This means that only "leaf" menu items without submenus have a true value.

As mentioned in Section 12.1, *Basic Menu Usage*, `menu_show()` by default returns the value of the item the user has selected. Since menu items are counted starting from one, a return value of zero from `menu_show()` would represent the null selection.⁷⁷ However, you may explicitly set the value of a menu item to zero. If you do, then a return value of zero could represent either a legal value for the selected item or an error. To tell whether or not the result was valid, call `menu_get()` with the boolean `MENU_VALID_RESULT`. A return value of `TRUE` means that the result was valid; `FALSE` means that the value is invalid.

Item Generate Procedures

As with the menu as a whole, you may specify a *generate procedure* for each menu item, to be called just before the item is displayed.

Item Action Procedures

The *action procedure* of a menu item is analogous to the notify procedure of a menu. This is your chance to do something immediately based on the user's selection.

Menu notify procedures and item action procedures differ in when they are called. If the user chooses an item in a pull-right menu, the notify procedures (if any) for the *menus* higher up in the chain leading to the pull-right will be called, whereas the *action* procedures (if any) for the chosen *menu item* and *menu items* under it ("to its right") will be called.⁷⁸

Client Data

Each menu item has a *client data* field, accessible through `MENU_CLIENT_DATA`, which is reserved for the application's use. You can use this attribute to associate a unique identifier, or a pointer to a private structure, with each menu item.

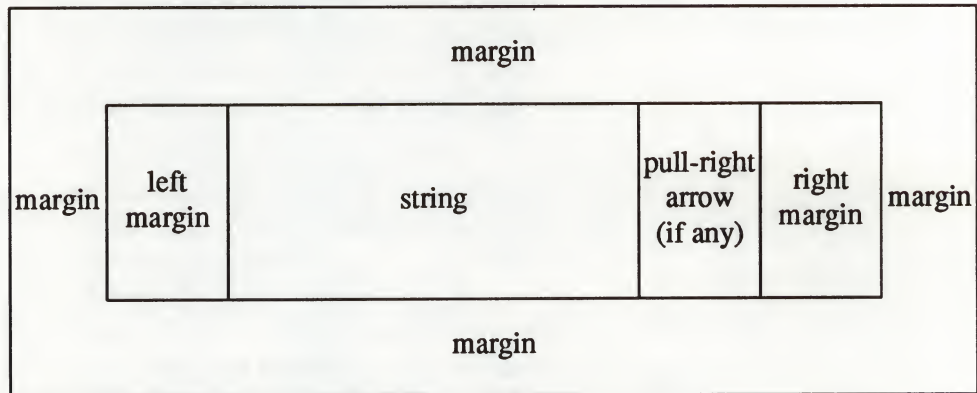
⁷⁷ This is why menu items are counted starting with one, rather than zero: so that a zero return value would represent the null selection whether the `menu_show()` was returning the value of the selected item or the item itself.

⁷⁸ Action procedures are discussed in detail in Section 12.7, *Callback Procedures*.

Item Margins

The diagram below illustrates the layout of a menu item:

Figure 12-1 *Layout of a Menu Item*



`MENU_MARGIN` represents the margin, in pixels, around an item in a menu. Its default value is 1.

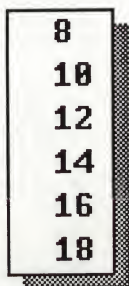
You can set an individual item's margin by setting the menu item. To set the margins for all items in a menu, set the menu's margin.

You can adjust the horizontal placement of text in menu items with `MENU_LEFT_MARGIN` and `MENU_RIGHT_MARGIN`.⁷⁹

As with `MENU_MARGIN`, the left and right margins can be set either for an individual menu item or for the menu itself, in which case the settings will apply to all the items in the menu. (The attributes `MENU_FONT` and `MENU_BOXED` also work this way.)

12.3. Examples

Example 4:



Our next example will show several variations on a simple menu that could be used for selecting font point sizes. The default form is shown to the left.

You could create the items with `MENU_STRINGS`, as in the previous example. Alternately, you could create the menu with no items, then use `menu_set()` to append the items to the menu:⁸⁰

```
m = menu_create(0);
for (i = 8; i <= 18; i += 2)
    menu_set(m, MENU_STRING_ITEM, int_to_str(i), i, 0);
```

⁷⁹ The placement of images is currently not affected by the settings of the left and right margins.

⁸⁰ Note that using `MENU_STRING_ITEM` with `menu_set()` has the effect of an implicit append. Several attributes are provided to explicitly add items to a menu — see Table 12-1, *Attributes to Add Pre-Existing Menu Items*, later in this section.

MENU_STRING_ITEM takes as values the item's string and its value.

Now let's see some of the ways in which the appearance of this basic menu can be altered.

By setting MENU_INACTIVE to TRUE for an item, you can "gray out" the item to indicate to the user that it is not currently selectable.

The menu to the left could be produced by:

```
for (i = 4; i <= 6; i++) {
    item = menu_get(m, MENU_NTH_ITEM, i);
    menu_set(item, MENU_INACTIVE, TRUE, 0);
}
```

Inactive items do not invert when the cursor passes over them.

The call `menu_set(m, MENU_BOXED, TRUE, 0)` will cause a single-pixel box to be drawn around each item. With the default margin of 1 pixel, this will result in two-pixel lines between each item.

Increasing the margin, by setting MENU_MARGIN to 5, will cause the items to spread out evenly, and the boxes to appear as individual boxes rather than dividing lines.

You can control the layout of the items within a menu with the attributes MENU_NCOLS and MENU_NROWS. Suppose you wanted the menu to be laid out horizontally instead of vertically:

8	10	12	14	16	18
---	----	----	----	----	----

All you need do is specify at create time that the menu will have 6 columns with a call such as `menu_set(m, MENU_NCOLS, 6, 0)`.

You can use MENU_NCOLS or MENU_NROWS to create two-dimensional menus, as well. The call `menu_set(m, MENU_NCOLS, 3, 0)` will cause the menu package to begin a second row after the first three columns have been filled with items:

8	10	12
14	16	18

The previous example specified that the menu have 3 columns. Specifying that it have 2 rows via `MENU_NROWS` would have the same effect. Items are laid out from upper left to lower right, in "reading order," regardless of how the layout is specified.

8	10	12
14	16	18

The only time you need to specify both the number of rows and the number of columns is when you want to fix the size of the menu, regardless of how many items it contains. Setting `MENU_NCOLS` to 3 and `MENU_NROWS` to 3 would produce:

If both dimensions of the menu are fixed and more items are given than will fit, the excess items will not appear.

You can remove the menu's shadow by setting `MENU_SHADOW` to null:

8
10
12
14
16
18

The menu package provides three predefined pixrects for the menu shadow. The call `menu_set(m, MENU_SHADOW, &menu_gray25_pr)` produces the 25 percent gray pattern shown on first menu below. Note that these are pixrects, *not* pixrect pointers. The other two patterns are produced by using `menu_gray50_pr` and `menu_gray75_pr`:

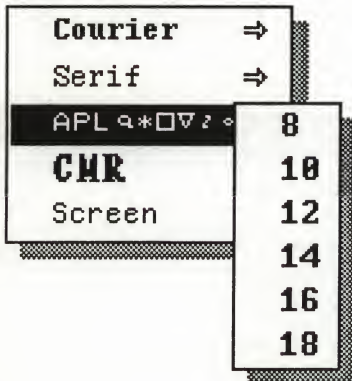
8
10
12
14
16
18

8
10
12
14
16
18

8
10
12
14
16
18

Example 5:

Let's take the size menu from the previous example and use it to create the more complex menu shown below, which the user could use to select both a font family and a point size within the family. This illustrates the multiple usage of a single menu. Pulling right over any of the items in the family menu will bring up the menu for selecting point size, as shown on the left.



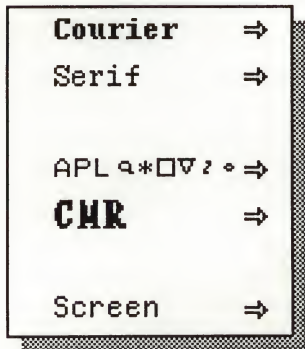
By using `MENU_ITEM`, we can give each item in the font family menu its string, the font in which to render the string, and the size menu as a pull-right:

```
family_menu = menu_create(
    MENU_ITEM,
        MENU_STRING,    "Courier",    MENU_FONT, cour,
        MENU_PULLRIGHT, size_menu,    0,
    MENU_ITEM,
        MENU_STRING,    "Serif",      MENU_FONT, serif,
        MENU_PULLRIGHT, size_menu,    0,
    MENU_ITEM,
        MENU_STRING,    "aplAPLGIJ",  MENU_FONT, apl,
        MENU_PULLRIGHT, size_menu,    0,
    MENU_ITEM,
        MENU_STRING,    "CMR",        MENU_FONT, cmr,
        MENU_PULLRIGHT, size_menu,    0,
    MENU_ITEM,
        MENU_STRING,    "Screen",     MENU_FONT, screen,
        MENU_PULLRIGHT, size_menu,    0,
    0);
```

Suppose the font family menu had already been created, and we wanted to add the size menu as a pull-right to each item of the existing menu. We could do this using the attributes `MENU_NITEMS` and `MENU_NTH_ITEM`. The loop below iterates over each item in the menu, retrieving the item's handle and setting the pull-right for the item:

```
for (i = (int)menu_get(family_menu, MENU_NITEMS); i > 0; --i)
    menu_set(menu_get(family_menu, MENU_NTH_ITEM, i),
        MENU_PULLRIGHT, size_menu, 0);
```


Example 6:



You can insert new items into an existing menu with `MENU_INSERT`. For example, suppose you want to insert blank lines into the font family menu, to indicate grouping:

You can do this by inserting non-selectable items into the menu:

```
menu_set(family_menu,
        MENU_INSERT,
        2,
        menu_create_item(MENU_STRING, "",
                        MENU_FEEDBACK, FALSE,
                        0),
0);

menu_set(family_menu,
        MENU_INSERT,
        5,
        menu_get(family_menu, MENU_NTH_ITEM, 3),
0);
```

`MENU_INSERT` takes two values: the number of the item to insert after, and the new item to insert. Disabling `MENU_FEEDBACK` makes the item non-selectable.

The above example uses `menu_create_item()` to explicitly create the item to be inserted. Usually menu items are created implicitly, using the attributes described in Table 12-2, *Menu Item Creation Attributes*, in the next section.

NOTE `menu_create_item()` does *not* set the `MENU_RELEASE` attribute by default, so that the resulting item will not be automatically destroyed when its parent menu is destroyed. This is in contrast to implicitly created menu items — see Section 12.5, *Destroying Menus*.

In addition to MENU_INSERT, there are several other attributes you can use to add pre-existing menu items to a menu.⁸¹ They are summarized in the following table.

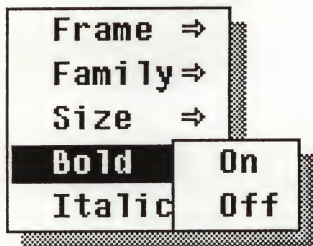
Table 12-1 *Attributes to Add Pre-Existing Menu Items*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
MENU_APPEND_ITEM	Menu_item	Append item to end of menu.
MENU_INSERT	int, Menu_item	Insert new item after nth item (use n=0 to prepend).
MENU_INSERT_ITEM	Menu_item(old), Menu_item(new)	Insert new item after old item.
MENU_REPLACE	int, Menu_item	Replace nth item with specified item.
MENU_REPLACE_ITEM	Menu_item(old), Menu_item(new)	Replace old item with new item in the menu (old item is not replaced in any other menus it may appear in).

⁸¹ To delete items from a menu, use MENU_REMOVE or MENU_REMOVE_ITEM, described in the *Menu Attributes* table in Chapter 19, *SunView Interface Summary*.

Example 7:

For the next example we will attach the on-off, family and size menus of the previous examples as pull-rights to a higher-level menu for selecting fonts:



```
font_menu = menu_create(
    MENU_PULLRIGHT_ITEM, "Frame",  frame_menu,
    MENU_PULLRIGHT_ITEM, "Family", family_menu,
    MENU_PULLRIGHT_ITEM, "Size",   size_menu,
    MENU_PULLRIGHT_ITEM, "Bold",   on_off_menu,
    MENU_PULLRIGHT_ITEM, "Italic", on_off_menu,
    0);
```

MENU_PULLRIGHT_ITEM takes a string and a menu as values. It creates an item represented by the string and with the menu as a pull-right.

Note that `on_off_menu` is used as a pull-right for both the bold and the italic menu items, and that the `size_menu` appears both as a pull-right from main level `font_menu` and from each item in `family_menu`. This demonstrates that a menu may have more than one parent. However, recursive menus are not allowed — if M1 is a parent of M2, M2 (or any of its children) may not have M1 as a child. Displaying such a recursive menu will probably result in a segmentation fault.

The 'Frame' item takes as its pull-right the menu which has been retrieved from the frame using `WIN_MENU`.

The program `font_menu`, printed in Appendix A, *Example Programs*, builds further on the above examples.

12.4. Item Creation Attributes

The attribute `MENU_ITEM`, introduced in Example 2, suffices to create any type of menu item. However, several attributes are provided for convenience as a shorthand way to create items with common attributes. These attributes, along with the types of values they take and the type of item they create, are summarized in the following table:

Table 12-2 *Menu Item Creation Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Type of Item Created</i>
<code>MENU_ACTION_IMAGE</code>	image, action proc	Image item w/action proc.
<code>MENU_ACTION_ITEM</code>	char *, action proc	String item w/action proc.
<code>MENU_GEN_PULLRIGHT_IMAGE</code>	Pixrect *, proc	Image item with generate proc for pull-right.
<code>MENU_GEN_PULLRIGHT_ITEM</code>	char *, proc	String item with generate proc for pull-right.
<code>MENU_IMAGE_ITEM</code>	Pixrect *, value	Image item w/value.
<code>MENU_IMAGES</code>	list of Pixrect *	Multiple image items.
<code>MENU_PULLRIGHT_IMAGE</code>	Pixrect *, Menu	Image item w/pull-right.
<code>MENU_PULLRIGHT_ITEM</code>	char *, Menu	String item w/pull-right.
<code>MENU_STRING_ITEM</code>	char *, value	String item w/value.
<code>MENU_STRINGS</code>	list of char *	Multiple string items.

We could now create the menu in Example 2 more compactly by using `MENU_PULLRIGHT_ITEM` instead of `MENU_ITEM`:

```
m = menu_create(MENU_PULLRIGHT_ITEM, "Bold", on_off_menu,
                MENU_PULLRIGHT_ITEM, "Italic", on_off_menu,
                0),
```


12.5. Destroying Menus

Both menus and menu items are destroyed with the function:

```
void
menu_destroy(menu_object)
    <Menu or Menu_item> menu_object;
```

CAUTION

Watch out for dangling pointers when using a menu item in multiple menus. The attribute `MENU_RELEASE` (which takes no value) controls whether or not a menu item is automatically destroyed when its parent menu is destroyed. `MENU_RELEASE` is set to `TRUE` by default for menu items created in-line via the menu item creation attributes. This can lead to dangling pointers, if the same menu item appears multiple times, because calling `menu_destroy()` can lead to items being destroyed multiple times. This warning also applies to pull-rights which are used multiple times. To prevent this error, remove multiple occurrences of an item or pull-right before destroying a menu.

Calling `menu_destroy_with_proc()` instead of `menu_destroy()` when you want to destroy a menu lets you specify a procedure to be called as the menu or menu item is destroyed. lets you specify a procedure to be called every time a particular menu or menu item is about to be destroyed:

```
void
menu_destroy_with_proc(menu_object, destroy_proc)
    <Menu or Menu_item> menu_object;
void (*destroy_proc)();
```

Your destroy procedure should be of the form:

```
void
destroy_proc(menu_object, type)
    <Menu or Menu_item> menu_object;
Menu_attribute type;
```

For menus, `menu_object` is the menu and the type parameter is `MENU_MENU`; for menu items, `menu_object` is the item and the type parameter is `MENU_ITEM`.

12.6. Searching for a Menu Item

The function `menu_find()` lets you search through a menu (and its children) to find a menu item meeting certain criteria:

```
Menu_item
menu_find(menu, attributes);
      Menu      menu;
      <attribute-list> attributes;
```

For example, the following call searches for the menu item whose string was "Load New File". `menu_find()` will return `itNULLif`

whose string was "Load New File":

By default, `menu_find()` uses a "deferred" search — searching all the items in a menu before descending into any pull-rights which may be present. By setting `MENU_DESCEND_FIRST` (which takes no value), you can force a depth-first search.

If multiple attributes are given, `menu_find()` will find the first item matching all the attributes.

The following attributes are recognized by `menu_find()`:

Table 12-3 *Menu Attributes Recognized by menu_find()*

MENU_ACTION	MENU_INVERT
MENU_CLIENT_DATA	MENU_LEFT_MARGIN
MENU_FEEDBACK	MENU_MARGIN
MENU_FONT	MENU_PARENT
MENU_GEN_PROC	MENU_PULLRIGHT
MENU_GEN_PULLRIGHT	MENU_RIGHT_MARGIN
MENU_IMAGE	MENU_STRING
MENU_INACTIVE	MENU_VALUE

12.7. Callback Procedures

When you call `menu_show()`, the menu package displays the menu, gets a selection from the user, and undisplay the menu. The menu package allows you to specify *callback procedures* which will be called at various points during the invocation of the menu. These let you create and modify menus or respond to the user's actions, on the fly, at the time the user brings up the menu. There are three types of callback procedures: *generate procedures* (so named because they are called before the menu or item is displayed, allowing the application to *generate* or modify the menu on the fly), *notify procedures* (for menus) and *action procedures* (for menu items) which are called after the user has made a selection.

Flow of Control in `menu_show()`

The callback mechanism gives you a great deal of flexibility in creating, combining and modifying menus and menu items. This flexibility comes at the price of some complexity, however. To take advantage of it, it is necessary to understand when the callback procedures are called after you invoke `menu_show()`.

For purposes of explanation, the diagrams below divide the process of displaying a menu and getting the user's selection into two stages, the *display stage* and the *notification stage*.

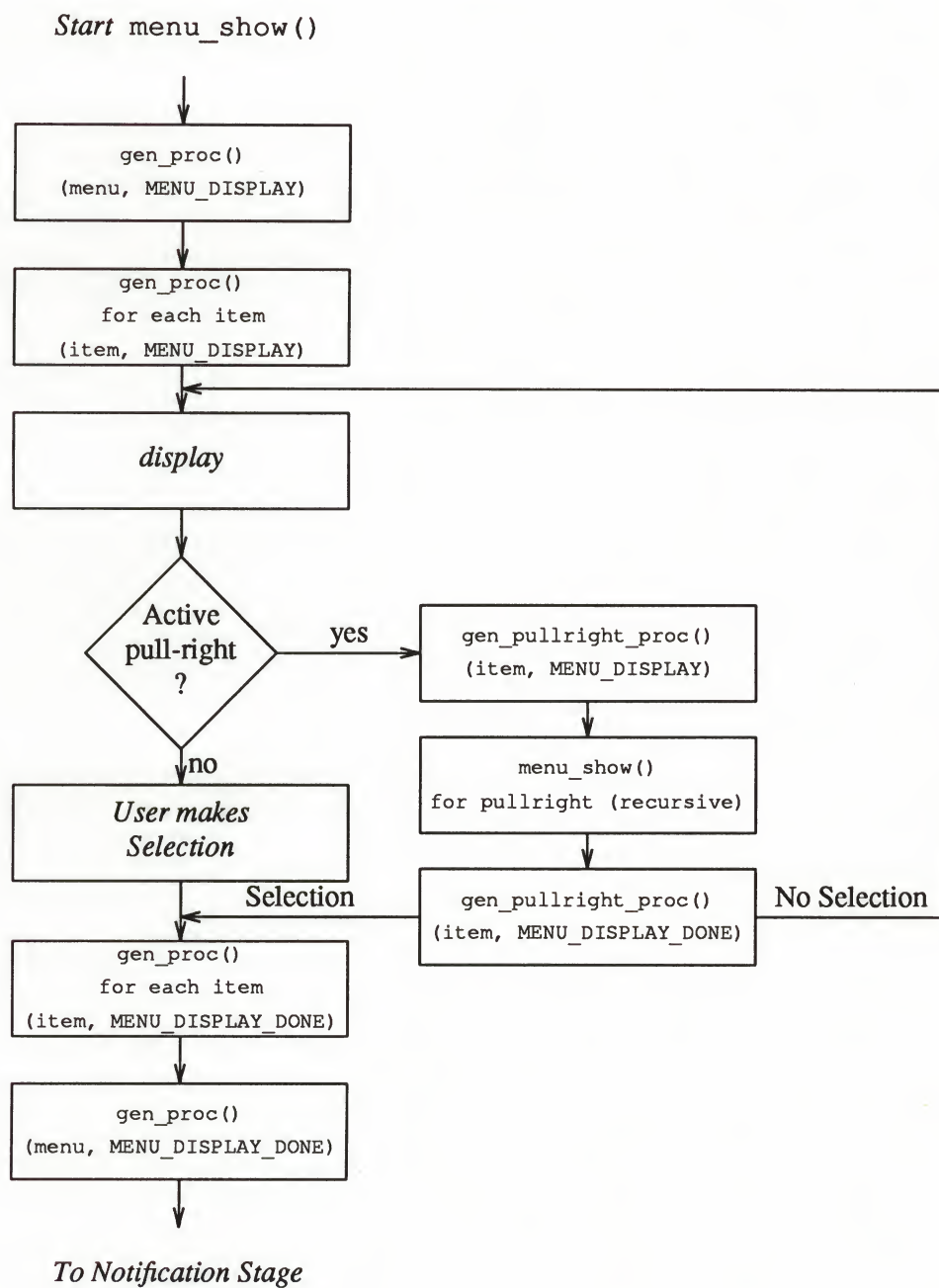
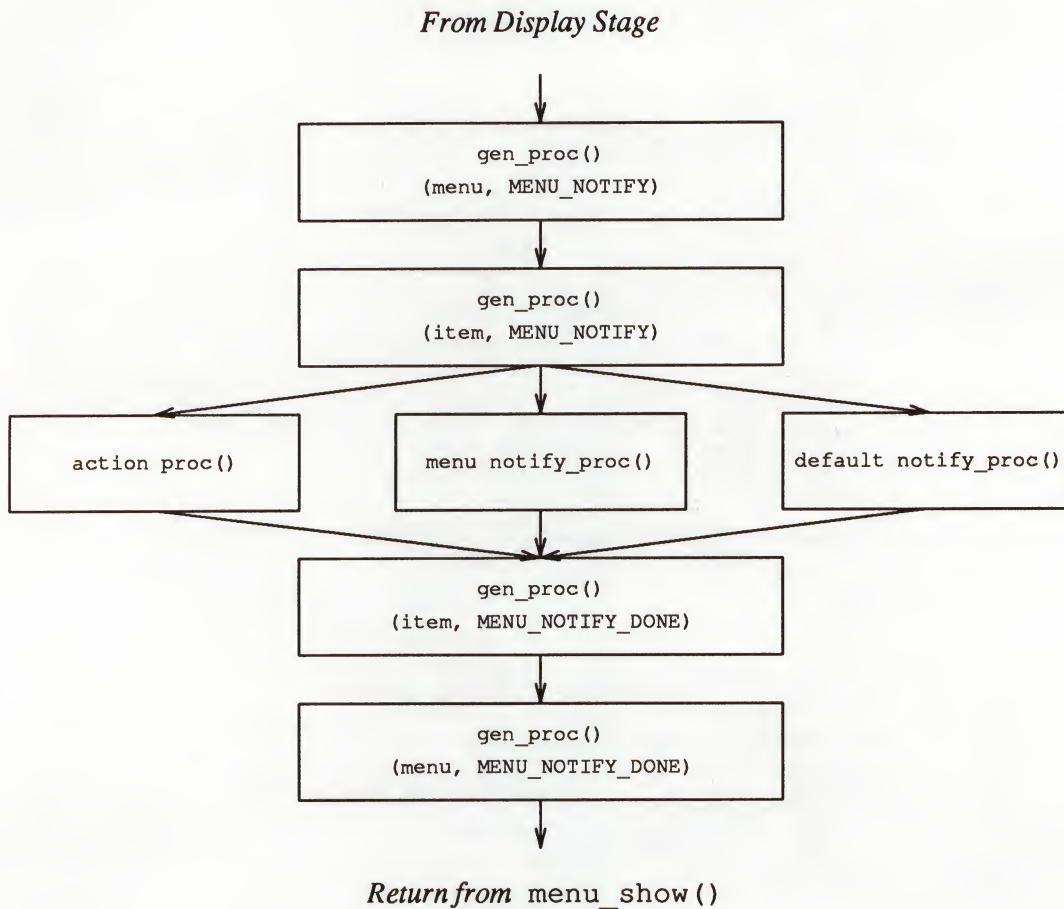
Figure 12-2 *Display Stage of Menu Processing*

Figure 12-3 Notification Stage of Menu Processing



Generate Procedures

The first argument to a generate procedure is either a menu or menu item depending on whether it's a `MENU_GEN_PROC` or a `MENU_GEN_PROC_ITEM`. Also passed in is an *operation* indicating at which point in the processing of the menu the generate procedure is being called. The operation parameter is of type `Menu_generate`, and may be `MENU_DISPLAY`, `MENU_DISPLAY_DONE`, `MENU_NOTIFY` or `MENU_NOTIFY_DONE`.⁸²

NOTE *The menu package uses the fullscreen access mechanism when displaying the menu. Writing to the screen while under fullscreen access will probably cause your program to deadlock, so your generate procedure should not access the screen when called with an operation of `MENU_DISPLAY` or `MENU_DISPLAY_DONE`.*

⁸² For a detailed explanation of when the generate procedures are called in relation to the other callback procedures, see the diagrams in the next subsection, *Flow of Control in menu_show()*.

There are three types of generate procedures — *menu item generate procedures*, *menu generate procedures*, and *pull-right generate procedures*. A description and example of each is given below.

Menu Item Generate Procedure

A generate procedure attached to a menu item has the form:

```
Menu_item
menu_item_gen_proc(item, operation)
    Menu_item    item;
    Menu_generate operation;
```

You can specify a menu item generate procedure via MENU_GEN_PROC.

Example 8:

The most common use of menu item generate procedures is to modify the item's display string. The program listed below registers a generate procedure, `toggle_proc()`. If it has been called from the MENU_DISPLAY stage of processing, it toggles the text of the 'Redisplay' item on the frame menu.

```
#include <suntool/sunview.h>

Menu_item toggle_proc();
int toggle = 0;

main()
{
    Window frame = window_create(NULL, FRAME, 0);
    Menu menu = window_get(frame, WIN_MENU);
    Menu_item item = menu_find(menu,
                                MENU_STRING, "Redisplay", 0);

    menu_set(item, MENU_GEN_PROC, toggle_proc, 0);
    window_main_loop(frame);
}

Menu_item
toggle_proc(mi, op)
    Menu_item    mi;
    Menu_generate op;
{
    switch (op) {
        case MENU_DISPLAY:
            if (toggle) {
                menu_set(mi,
                        MENU_STRING, "Redisplay has been seen",
                        0);
            } else {
                menu_set(mi,
                        MENU_STRING, "Redisplay",
                        0);
            }
            toggle = !toggle;
            break;

        case MENU_DISPLAY_DONE:
        case MENU_NOTIFY:
```



```

        case MENU_NOTIFY_DONE:
            break;
    }
    return mi; /* item handle always returned */
}

```

The 'Zoom'/'Unzoom' item in the SunView frame menu also uses this technique to toggle its display string. Note that since this item knows how to modify itself, you could put it in other menus and get the same behavior. A generate procedure for a menu item allows the application to be called even when it has no knowledge of or control over the call to `menu_show()`.

Menu Generate Procedure

A generate procedure attached to a menu has the form:

```

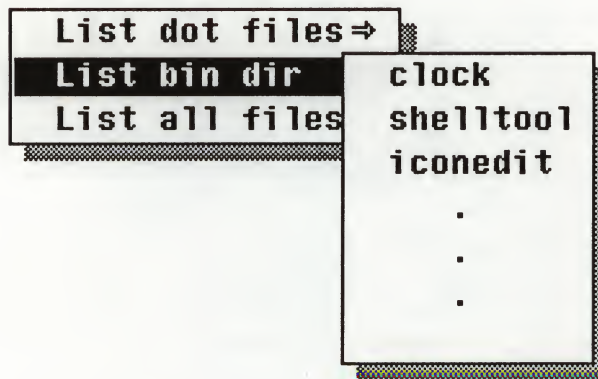
Menu
menu_gen_proc(m, operation)
    Menu      m;
    Menu_generate operation;

```

You can specify a menu generate procedure via the attribute `MENU_GEN_PROC`.

Example 9:

We will take as an example a menu allowing the user to list different groups of files. When the user makes a selection, we generate a menu containing the correct set of files:



The relevant functions are listed on the next page. The first, `initialize_menu()`, creates the three menu items, giving each of them the generate procedure `list_files()`, and a unique identifier as `MENU_CLIENT_DATA`.

Remember that `list_files()` is called in four different situations by `menu_show()`:⁸³

- When the operation is `MENU_DISPLAY`, the pull-right is being asked to display its menu, so `list_files()` calls the function `get_file_names()` (not shown) to get the appropriate list of file names,

⁸³ See the diagrams in the earlier subsection, *Flow of Control in menu_show()*.

and adds each name in the list to the menu.

- When `list_files()` is called with operation set to `MENU_DISPLAY_DONE`, the menu of generated file names is no longer being displayed. `list_files()` cleans up by destroying the old menu of file names, replacing it with a fresh menu with the same generate procedure. It returns the handle of this new menu.
- When `list_files()` is called with an operation of `MENU_NOTIFY` or `MENU_NOTIFY_DONE`, the menu is returned unaltered.

```
#define DOT 0
#define BIN 1
#define ALL 2

static void
initialize_menu(menu)
    Menu menu;
{
    m = menu_create(MENU_GEN_PROC, list_files,
                   MENU_CLIENT_DATA, DOT,
                   0);
    menu_set(menu,
             MENU_PULLRIGHT_ITEM, "List dot files", m,
             0);
    m = menu_create(MENU_GEN_PROC, list_files,
                   MENU_CLIENT_DATA, BIN,
                   0);
    menu_set(menu,
             MENU_PULLRIGHT_ITEM, "List bin dir", m,
             0);
    m = menu_create(MENU_GEN_PROC, list_files,
                   MENU_CLIENT_DATA, ALL,
                   0);
    menu_set(menu,
             MENU_PULLRIGHT_ITEM, "List all files", m,
             0);
}

static Menu
list_files(m, operation)
    Menu m;
    Menu_generate operation;
{
    char **list;
    int directory;
    int i = 0;

    switch (operation) {
        case MENU_DISPLAY:
            directory = (int)menu_get(m, MENU_CLIENT_DATA);
            list = get_file_names(directory);
            while (*list)
                menu_set(m,
                        MENU_STRING_ITEM, *list++, i++,
                        0);
            break;
    }
}
```



```

        case MENU_DISPLAY_DONE:
            /*
             * Destroy old menu and all its entries.
             * Replace it with a new menu.
             */
            directory = (int)menu_get(m, MENU_CLIENT_DATA);
            menu_destroy(m);
            m = menu_create(MENU_GEN_PROC,    list_files,
                           MENU_CLIENT_DATA, directory,
                           0);

            break;

        case MENU_NOTIFY:
        case MENU_NOTIFY_DONE:
            break;
    }

    /* The current or newly-created menu is returned */
    return m;
}

```

Pull-right Generate Procedure

You can postpone the generation of a pull-right menu until the user actually pulls right by specifying a pull-right generate procedure. A pull-right generate procedure has the form:

```

Menu
pullright_gen_proc(mi, operation)
    Menu_item    mi;
    Menu_generate operation;

```

Note that the pull-right generate procedure is passed the item, and returns the menu to be displayed.

You can specify a menu item's pull-right generate procedure with a call such as

```
menu_set(menu_item, MENU_GEN_PULLRIGHT, my_pullright_gen, 0);
```

Alternatively, you can use the attributes `MENU_GEN_PULLRIGHT_IMAGE` or `MENU_GEN_PULLRIGHT_ITEM` to give a menu both an item and the item's generate procedure.

If you want to get the existing menu for an item which has a pull-right generate procedure, retrieve the value of the item, as in:

```
menu = menu_get(item, MENU_VALUE);
```

Notify/Action Procedures

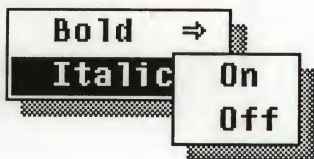
When the user selects a menu item by releasing the mouse button, the menu package calls back to any notify procedures or action procedures you have specified. Notify procedures and action procedures have the form:

```
caddr_t
notify_proc(m, mi)
    Menu      m;
    Menu_item mi;
```

The most common usage is to have action procedures for the items at the leaf nodes of the walking menu. The general mechanism described below is provided to allow your procedures to be called for non-leaf nodes as well.

Imagine a chain of menus expanded out. Lookup of the notify/action procedures starts with the “oldest” menu, the one passed to `menu_show()`. If it has a notify procedure, that notify procedure is called, otherwise the default notify procedure, `menu_return_value()`, is called. Likewise, for each menu down the chain, until the menu with the selected item is reached. If the selected item has an action procedure, that action procedure is called. If the selected item is not on a leaf node, then action procedures for any items farther down the chain are also called.

Let's see what happens in the example to the left (assume that 'On' is the default item for the first menu):



If 'Italic' was selected:

- no callback to the first menu's notify procedure since an item in it is selected,
- callback to the action procedure for the 'Italic' item,
- no callback to the second menu's notify procedure since it is further down the chain than the selected item,
- callback to the action procedure for the 'On' item.

If 'Off' was selected:

- callback to the notify procedure for the first menu, since an item in a menu further down the chain than it is selected,
- no callback to the action procedure for the 'Italic' item, since it is above the selected item in the chain,
- no callback to the second menu's notify procedure since an item in it is selected,
- callback to the action procedure for the 'Off' item.

NOTE *If you specify a notify procedure, it is your responsibility to propagate the notification to any menus further down in the chain. You can do this by calling `menu_get(mi, MENU_VALUE)` from your notify procedure. This gets the value of the selected menu item, and since the value of a pull-right item is the*

value of its pull-right menu, this will make notify/action procedures further down the chain get called.

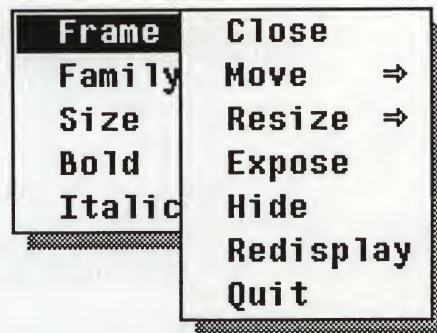
12.8. Interaction with Previously Defined SunView Menus

Walking Menus for frames and tty subwindows can be customized.⁸⁴ All menu items in these menus are “position-independent” — in other words the menus do not count on a given item having a certain position or being located in a particular menu. This makes it possible for you to safely add new items (including pull-right submenus) to an existing menu.

NOTE You should not use the client data field of items created by SunView packages, because the packages have pre-empted it for their own use.

Using an Existing Menu as a Pull-right

The program *font_menu*, listed in Appendix A, shows how you can replace an existing menu with your own menu which has the original menu as a pull-right. Making use of several of the examples given earlier in the chapter, it creates a font menu which allows the user to select the font family, point size, and whether or not the font is bold or italic. Meanwhile, the first item, labelled ‘Frame’, brings up the original frame menu:



⁸⁴ Remember that in order to have these packages use walking menus the user must have enabled the *Walking_Menus* option in *SunView* category of `defaultsedit(1)`; in SunOS Release 4.0, this is the default.

12.9. Initial and Default Selections

Two special menu items are the *default item* (MENU_DEFAULT_ITEM) and the *selected item* (MENU_SELECTED_ITEM). The default item is simply a distinguished item. The selected item is the item which was last selected.

Two attributes are provided to control the behavior of a menu in regard to its initial selection. If MENU_INITIAL_SELECTION_SELECTED is TRUE, the menu comes up with its initial selection selected — that is the selection is inverted and the cursor is positioned over it. If FALSE, the menu comes up with the cursor “standing off” to the left and no selection highlighted. If MENU_INITIAL_SELECTION_EXPANDED is TRUE, when the menu comes up, it automatically expands any pull-rights which are necessary to bring the initial selection up on the screen.

Each menu also has an *initial selection* (MENU_INITIAL_SELECTION) and a *default selection*. (MENU_DEFAULT_SELECTION).

The distinction between the initial selection and the default selection is subtle. Suppose MENU_INITIAL_SELECTION_EXPANDED was TRUE, and the initial selection was an item in a pull-right. When the menu comes up, it will be expanded to show the initial item as selected. However, if the user moves the cursor to the left, backing out of the pull-right, and then moves back to the right, bringing the pull-right up again, the item selected will be the default selection rather than the initial selection.

When the user selects a pull-right item without bringing up the associated menu, it is as if he had brought the pull-right up and selected the default item.

You can set the initial selection and the default selection independently — either can be set to the default item or the selected item.

12.10. User Customizable Attributes

The user can specify the values of certain menu attributes in the *Menu* category of `defaultsedit(1)`. When a menu is created, for attributes not explicitly specified by the application program, the menu package retrieves the values set by the user from the defaults database maintained by `defaultsedit`. This allows the user the ability to tailor, to some extent, the appearance and behavior of menus across different applications. For example, he may want to change the type of shadow, or expand the menu margin, and so on.

The attributes under `defaultsedit` control are listed in the following table.

Table 12-4 *User Customizable Menu Attributes*

<i>Attribute</i>	<i>Default</i>	<i>Description</i>
MENU_BOXED	FALSE	If TRUE, a single-pixel box will be drawn around each menu item.
MENU_DEFAULT_SELECTION	MENU_DEFAULT	MENU_SELECTED or MENU_DEFAULT.
MENU_FONT	screen.b.12	Menu's font.
MENU_INITIAL_SELECTION	MENU_DEFAULT	MENU_SELECTED or MENU_DEFAULT.
MENU_INITIAL_SELECTION_SELECTED	FALSE	If TRUE, menu comes up with its initial selection highlighted. If FALSE, menu comes up with the cursor "standing off" to the left.
MENU_INITIAL_SELECTION_EXPANDED	TRUE	If TRUE, when the menu pops up, it automatically expands to select the initial selection.
MENU_JUMP_AFTER_NO_SELECTION	FALSE	If TRUE, cursor jumps back to its original position after no selection made.
MENU_JUMP_AFTER_SELECTION	FALSE	If TRUE, cursor jumps back to its original position after selection made.
MENU_MARGIN	1	The margin around each item.
MENU_LEFT_MARGIN	16	For each string item, margin in addition to MENU_MARGIN on left between menu's border and text.
MENU_PULLRIGHT_DELTA	9999	# of pixels the user must move the cursor to the right to cause a pull-right menu to pop up.
MENU_RIGHT_MARGIN	6	For each string item, margin in addition to MENU_MARGIN on right between menu's border and text.
MENU_SHADOW	50% grey	Pattern for menu's shadow.

Cursors

Cursors	253
13.1. Creating and Modifying Cursors	255
13.2. Copying and Destroying Cursors	255
13.3. Crosshairs	256
13.4. Some Cursor Attributes	257



Cursors

This chapter describes how to create and manipulate *cursors*.⁸⁵ A cursor is an image that tracks the mouse on the display. Each window in SunView has its own cursor, which you can change with the cursor package.

If it is installed on your system, you can run the demo `/usr/demo/cursor_demo` to see the effects of various cursor attributes. The source for this is in `/usr/src/share/sun/suntool/cursor_demo.c`.

Header Files

The definitions necessary to use cursors are found in the include file `<sunwindow/win_cursor.h>`, which is included by default when you include the file `<suntool/sunview.h>`.

Summary Listing and Tables

To give you a feeling for what you can do with cursors, the following page contains a list of the available cursor attributes and functions. Many of these are discussed in the rest of this chapter and elsewhere (use the *Index* to check). All are briefly described with their arguments in the cursor summary tables in Chapter 19, *SunView Interface Summary*:

- the *Cursor Attributes* table begins on page 321;
- the *Cursor Functions* table begins on page 323.

⁸⁵ The cursor is called the “pointer” in user-level documentation.

Cursor Attributes

CURSOR_CROSSHAIR_BORDER_GRAVITY	CURSOR_OP
CURSOR_CROSSHAIR_COLOR	CURSOR_SHOW_CROSSHAIRS
CURSOR_CROSSHAIR_GAP	CURSOR_SHOW_CURSOR
CURSOR_CROSSHAIR_LENGTH	CURSOR_SHOW_HORIZ_HAIR
CURSOR_CROSSHAIR_OP	CURSOR_SHOW_VERT_HAIR
CURSOR_CROSSHAIR_THICKNESS	CURSOR_VERT_HAIR_BORDER_GRAVITY
CURSOR_FULLSCREEN	CURSOR_VERT_HAIR_COLOR
CURSOR_HORIZ_HAIR_BORDER_GRAVITY	CURSOR_VERT_HAIR_GAP
CURSOR_HORIZ_HAIR_COLOR	CURSOR_VERT_HAIR_LENGTH
CURSOR_HORIZ_HAIR_GAP	CURSOR_VERT_HAIR_OP
CURSOR_HORIZ_HAIR_LENGTH	CURSOR_VERT_HAIR_THICKNESS
CURSOR_HORIZ_HAIR_OP	CURSOR_XHOT
CURSOR_HORIZ_HAIR_THICKNESS	CURSOR_YHOT
CURSOR_IMAGE	

Cursor Functions

cursor_copy(src_cursor)	cursor_get(cursor, attribute)
cursor_create(attributes)	cursor_set(cursor, attributes)
cursor_destroy(cursor)	

13.1. Creating and Modifying Cursors

The basic usage of the cursor package is to first create a cursor with `cursor_create()`, and then use this cursor as the value of the `WIN_CURSOR` attribute in your call to `window_create()`.

```
Cursor
cursor_create(attributes)
    <attribute-list> attributes;
```

Once you have created a cursor, you can alter its attributes with `cursor_set()` and read back its attributes with `cursor_get()`:

```
void
cursor_set(cursor, attributes)
    Cursor cursor;
    <attribute-list> attributes;

caddr_t
cursor_get(cursor, attribute)
    Cursor cursor;
    Cursor_attribute attribute;
```

If you want to change the cursor of a window that has already been created, you can first get the cursor from the window using `window_get()` of `WIN_CURSOR`, then use `cursor_set()` to change the cursor, and then use `window_set()` of `WIN_CURSOR` to re-attach the cursor to the window.

13.2. Copying and Destroying Cursors

A copy of an existing cursor can be made with `cursor_copy()`:

```
Cursor
cursor_copy(src_cursor)
    Cursor src_cursor;
```

A cursor can be destroyed and its resources freed with `cursor_destroy()`:

```
void
cursor_destroy(cursor)
    Cursor cursor;
```

Example 1: Creating a Window with a Custom Cursor

A common use for cursors might be to create a canvas subwindow and have it use the cursor of your choice, rather than the default arrow cursor:


```

short my_pixrect_data[] = {
#include "file_from_iconedit"
};
mpr_static(my_pixrect, 16, 16, 1, my_pixrect_data);

Canvas canvas;

init_my_canvas()
{
    canvas = window_create(frame, CANVAS,
        WIN_CURSOR, cursor_create(CURSOR_IMAGE, &my_pixrect,
                                0),
        0);
}

```

This example creates a cursor “on the fly” and passes it into the `window_create()` routine for use with the canvas. The attribute `CURSOR_IMAGE` is set to the a pointer to the pixrect we want to use (a diamond or bullseye, for example). All of the other cursor attributes default to the value shown in the attribute table.

Example 2: Changing the Cursor of an Existing Window

Suppose you have already created a window and you want to change its cursor. Let's say you want to change the drawing op to `PIX_SRC`:

```

Cursor cursor;

cursor = window_get(my_window, WIN_CURSOR);
cursor_set(cursor, CURSOR_OP, PIX_SRC, 0);
window_set(my_window, WIN_CURSOR, cursor, 0);

```

CAUTION The cursor returned by `window_get()` is a pointer to a static cursor that is shared by all the windows in your application. So, for example, saving the cursor returned by `window_get()` and then making other window system calls might result in the saved cursor being overwritten.⁸⁶

It is safe to get the cursor, modify it with `cursor_set()` and then put the cursor back. If there is any chance that the static cursor will be overwritten, you should use `cursor_copy()` to make a copy of the cursor, then use `cursor_destroy()` when you are done.

13.3. Crosshairs

Crosshairs are horizontal and vertical lines whose intersection tracks the location of the mouse. You can control the appearance of both the horizontal and vertical crosshairs along with the cursor image. For example, you can create a cursor that only shows the cursor image, or only the horizontal crosshair, or both the horizontal and vertical crosshairs and the cursor image. By default both the crosshairs are turned off and only the cursor image is displayed.

⁸⁶ Note that this would happen if one of the routines you call happens to call `window_get()` of `WIN_CURSOR`.

Example 3: Turning on the Crosshairs

Suppose you have a canvas window in which you want to turn on both the horizontal and vertical crosshairs. This can be done by getting the cursor from the window and setting the `CURSOR_SHOW_CROSSHAIRS` attribute:

```
Cursor cursor;

cursor = window_get(my_canvas, WIN_CURSOR);
cursor_set(cursor, CURSOR_SHOW_CROSSHAIRS, TRUE, 0);
window_set(my_canvas, WIN_CURSOR, cursor, 0);
```

When the crosshairs are turned on, they are displayed according to the current value of their other attributes (e.g. thickness and drawing op).

13.4. Some Cursor Attributes

This section describes some of the cursor attributes in more detail. Note that for the crosshair attributes, you can control the individual crosshairs as well as both crosshairs by using the appropriate attribute. For example, you can set the length for both crosshairs with `CURSOR_CROSSHAIR_LENGTH` or the length of only the horizontal crosshair with `CURSOR_HORIZ_HAIR_LENGTH`.

`CURSOR_IMAGE`

The cursor image is the memory pixrect that is drawn on the screen as the mouse moves. Use the `mpr_static()` macro, as shown in Example 1, to create the memory pixrect. The image is represented as an array of 16 shorts, each of which represents a 16-pixel wide scan line. The scan lines are usually arranged in a single column, yielding a 16 x 16 pixel image. Other arrangements, such as 32 pixels wide x 8 pixels deep, are also possible. The maximum size of a cursor in SunView 1 is 32 bytes; the minimum width is 16, the width of one scan line.

`CURSOR_XHOT` and `CURSOR_YHOT`

The “hot spot” defined by (`CURSOR_XHOT`, `CURSOR_YHOT`) associates the cursor image, which has height and width, with the mouse position, which is a single point on the screen. The hot spot gives the mouse position an offset from the upper-left corner of the cursor image. For example, if the upper left corner of the cursor image is at location (50, 40) and the cursor hot spot has been set to (8, 8), the reported mouse position will be at (58, 48).

Most cursors have a hot spot whose position is obvious from the image shape: the tip of an arrow, the center of a bullseye, the center of a cross-hair. Cursors can also be used to give status feedback — an hourglass to indicate that the program is not responding to user input is a typical example. This type of cursor should have the hot spot located in the middle of its image so the user has a definite spot for pointing and does not have to guess where the hot spot is.

`CURSOR_OP`

The value given for this attribute is the rasterop which will be used to paint the cursor.⁸⁷ `PIX_SRC | PIX_DST` is generally effective on light backgrounds — in text, for example — but invisible over solid black. `PIX_SRC ^ PIX_DST` is a reasonable compromise over many different backgrounds, although it does poorly over a gray pattern.

⁸⁷ Rasterops are described fully in the *Pixrect Reference Manual*.

CURSOR_FULLSCREEN

The cursor crosshairs can be clipped to either the cursor's window or the entire screen. If you want the crosshairs to extend past the edge of the window, set **CURSOR_FULLSCREEN** to **TRUE**.

CURSOR_CROSSHAIR_LENGTH

If you don't want the crosshairs to cover the entire window (or screen), you can set the length of both crosshairs with **CURSOR_CROSSHAIR_LENGTH**. The value of this attribute is actually half the total crosshair length. For example, if you want the crosshairs to be 400 pixels wide and high, set the **CURSOR_CROSSHAIR_LENGTH** to 200. You can restore the extend-to-edge length by giving a value of **CURSOR_TO_EDGE** for **CURSOR_CROSSHAIR_LENGTH**.

CURSOR_CROSSHAIR_BORDER_GRAVITY

If the crosshair border gravity is enabled, the crosshairs will "stick" to the edge of the window (or screen). This is only interesting if the **CURSOR_CROSSHAIR_LENGTH** is not set to **CURSOR_TO_EDGE**. With border gravity turned on, each half of each crosshair will be attached to the edge of the window. With the cursor image displayed, this feature might be useful to help the user line up the cursor to a grid drawn on the edges of the window.

CURSOR_CROSSHAIR_GAP

If you don't want the halves of each crosshair to touch, you can set the **CURSOR_CROSSHAIR_GAP** to the half-length of space to leave between each crosshair half. If you set **CURSOR_CROSSHAIR_GAP** to **CURSOR_TO_EDGE**, the crosshairs will back off to the edge of the **CURSOR_IMAGE** rectangle.

Icons

Icons	261
14.1. Using Images Generated With <code>iconedit</code>	262
14.2. Modifying the Icon's Image	263
14.3. Loading Icon Images At Run Time	263



Icons

An *icon* is a small (usually 64 by 64 pixel) picture representing a base frame in its closed state. The icon is typically a picture indicating the function of the underlying application.

Header Files

The definitions necessary to use icons are found in the file `<suntool/icon.h>`, which is included by default when you include the file `<suntool/sunview.h>`.

Summary Listing and Tables

To give you a feeling for what you can do with icons, the following page lists the available icon attributes, functions and macros. Many of these are discussed in the rest of this chapter and elsewhere (use the *Index* to check). All are briefly described with their arguments in the menu summary tables in Chapter 19, *Sun-View Interface Summary*:

- the *Icon Attributes* table begins on page 328;
- the *Icon Functions and Macros* table begins on page 329.

<i>Icon Attributes</i>		
ICON_FONT	ICON_IMAGE_RECT	ICON_WIDTH
ICON_HEIGHT	ICON_LABEL	
ICON_IMAGE	ICON_LABEL_RECT	

<i>Icon Functions and Attributes</i>	
icon_create(attributes)	icon_set(icon, attributes)
icon_destroy(icon)	DEFINE_ICON_FROM_IMAGE(name, image)
icon_get(icon, attribute)	

14.1. Using Images Generated With iconedit

You can create and edit images easily using the program `iconedit(1)`. The output of `iconedit` is a file containing an array of shorts representing the image. In order to use the image in a program, you must first define a static memory `pixrect` containing this data. The `mpr_static()` macro is provided for this purpose.

The first argument to `mpr_static()` is the name of the `pixrect` to be defined. Next come the width, height and depth of the image, typically 64, 64 and 1. The last argument is the array of shorts containing the bit pattern of the icon image. For example:

```
static short icon_image[] = {
#include "file_generated_by_iconedit"
};
mpr_static(icon_pixrect, 64, 64, 1, icon_image);
```

The statically defined image is passed in to `icon_create()` at run time:

```
my_icon = icon_create(ICON_IMAGE, &icon_pixrect, 0);
```

Once you have created an icon, you can retrieve and modify its attributes with `icon_get()` and `icon_set()`, and destroy it with `icon_destroy()`.

Instead of creating the icon dynamically with `icon_create()`, you can use the `DEFINE_ICON_FROM_IMAGE()` macro to generate a static icon.⁸⁸

```
static short icon_image[] = {
#include "file_generated_by_iconedit"
};
DEFINE_ICON_FROM_IMAGE(icon, icon_image);
```

This macro statically allocates a structure representing an icon. Note that you

⁸⁸ The structure generated is actually an extern.

must pass the *address* of this structure — `&icon` in the example above — into `icon_get()`, `icon_set()`, and `icon_destroy()`.

WARNING *The `DEFINE_ICON_FROM_IMAGE()` macro may not be supported in future releases. We recommend that you use `icon_create()` instead.*

14.2. Modifying the Icon's Image

It is often useful to change the icon's image dynamically, rather than simply using the icon as a static placeholder. When *mailtool* receives new mail, for example, it lets the user know by modifying its icon to show a letter arrived in the mailbox. *clocktool* uses its icon to represent a moving clock face.

The steps to follow in modifying an icon's image are:

- get the frame's icon (attribute `FRAME_ICON`);
- get the icon's pixrect (attribute `ICON_IMAGE`);
- modify the pixrect as desired, or substitute a new pixrect;
- give the pixrect with the new image back to the icon;
- give the new icon back to the frame.

For example:

```
modify_icon(frame);
    Frame frame;

    Icon icon;
    Pixrect *pr;

    icon = (Icon) window_get(frame, FRAME_ICON);
    pr = (Pixrect *) icon_get(icon, ICON_IMAGE);
    ...
    (modify pr)
    ...
    icon_set(icon, ICON_IMAGE, pr, 0);
    window_set(frame, FRAME_ICON, icon, 0);
}
```

14.3. Loading Icon Images At Run Time

Often it is sufficient to define the image for a program's icon at compile time, with `mpr_static()`. However, you may want to allow the user to create his own icon images, and give the names of the files containing the images to your program as command-line arguments. Then you can load the images from the files the user has specified. Routines to load icon images from files at run time are described in Chapter 11 of the *SunView 1 System Programmer's Guide*.

Scrollbars

Scrollbars	267
15.1. Scrolling Model	269
15.2. Scrollbar User Interface	271
Types of Scrolling Motion	271
Undoing a Scroll	271
15.3. Creating, Destroying and Modifying Scrollbars	272
15.4. Programmatic Scrolling	275



Scrollbars

The canvas, text and panel subwindows have been designed to work with scrollbars. The text subwindow automatically creates its own vertical scrollbar. For canvases and panels, it is your responsibility to create the scrollbar and pass it in with the attributes `WIN_VERTICAL_SCROLLBAR` or `WIN_HORIZONTAL_SCROLLBAR`.

Section 15.2, *Scrollbar User Interface*, describes how the user interacts with scrollbars. Basic scrollbar usage is covered in Section 15.3, *Creating, Destroying and Modifying Scrollbars*, and programmatic scrolling is covered in Section 15.4, *Programmatic Scrolling*.

You may want to use scrollbars in an application not based on canvases, text subwindows or panels, in which case you must manage the interaction with the scrollbar directly. For an explanation of how to do this, see the *Scrollbars* chapter in the *SunView 1 System Programmer's Guide*.

Header Files

The definitions necessary to use scrollbars are found in the header file `<suntool/scrollbar.h>`

Summary Listing and Tables

To give you a feeling for what you can do with scrollbars, the following page contains a list of the available scrollbar attributes, functions and macros. Many of these are discussed in the rest of this chapter and elsewhere (use the *Index* to check). All are briefly described with their arguments in the scrollbar summary tables in Chapter 19, *SunView Interface Summary*:

- the *Scrollbar Attributes* table begins on page 362;
- the *Scrollbar Functions* table begins on page 365.

Scrollbar Attributes

SCROLL_ABSOLUTE_CURSOR	SCROLL_NOTIFY_CLIENT
SCROLL_ACTIVE_CURSOR	SCROLL_NORMALIZE
SCROLL_ADVANCED_MODE	SCROLL_OBJECT
SCROLL_BACKWARD_CURSOR	SCROLL_OBJECT_LENGTH
SCROLL_BAR_COLOR	SCROLL_PAGE_BUTTONS
SCROLL_BAR_DISPLAY_LEVEL	SCROLL_PAGE_BUTTON_LENGTH
SCROLL_BORDER	SCROLL_PAINT_BUTTONS_PROC
SCROLL_BUBBLE_COLOR	SCROLL_PIXWIN
SCROLL_BUBBLE_DISPLAY_LEVEL	SCROLL_PLACEMENT
SCROLL_BUBBLE_MARGIN	SCROLL_RECT
SCROLL_DIRECTION	SCROLL_REPEAT_TIME
SCROLL_END_POINT_AREA	SCROLL_REQUEST_MOTION
SCROLL_FORWARD_CURSOR	SCROLL_REQUEST_OFFSET
SCROLL_GAP	SCROLL_THICKNESS
SCROLL_HEIGHT	SCROLL_TO_GRID
SCROLL_LAST_VIEW_START	SCROLL_TOP
SCROLL_LEFT	SCROLL_VIEW_LENGTH
SCROLL_LINE_HEIGHT	SCROLL_VIEW_START
SCROLL_MARGIN	SCROLL_WIDTH
SCROLL_MARK	

Scrollbar Functions and Macros

scrollbar_create(attributes)	scrollbar_paint(scrollbar)
scrollbar_destroy(scrollbar)	scrollbar_paint_clear(scrollbar)
scrollbar_get(scrollbar, attribute)	scrollbar_clear_bubble(scrollbar)
scrollbar_set(scrollbar, attributes)	scrollbar_paint_bubble(scrollbar)
scrollbar_scroll_to(scrollbar, new_view_start)	

15.1. Scrolling Model

Scrollbars allow the user to control which portion of an object is visible when the object is larger than the window it is displayed in. Within the scrollbar is a darker area called the *bubble*. The size and position of the bubble within the bar tell the user where he is in the object and how much of the object is visible. By moving the bubble within the bar, the user brings different portions of the object into view.

The length of the object, the length of the visible portion of the object, and the offset of the visible portion within the object are given by the attributes `SCROLL_OBJECT_LENGTH`, `SCROLL_VIEW_LENGTH`, and `SCROLL_VIEW_START`. The relationship between these three view-space metrics is shown in the figure on the next page.

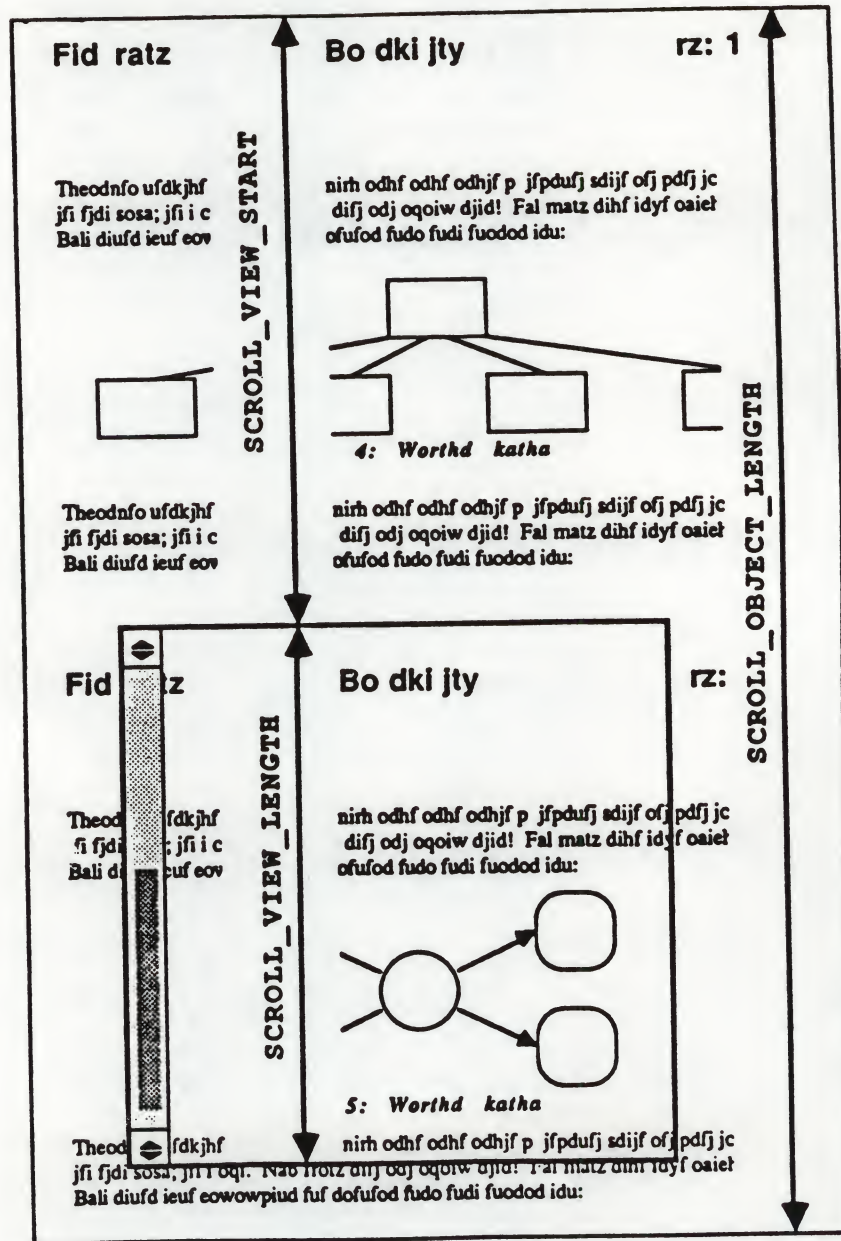
Figure 15-1 *Scrolling Model*

Figure 15-1 shows a two-page document being viewed within a window roughly half the size of the document. The three view-space attributes `SCROLL_OBJECT_LENGTH`, `SCROLL_VIEW_LENGTH`, and `SCROLL_VIEW_START` are shown superimposed on the document. Note the relative size and position of the bubble within the scrollbar — it is roughly half the size of the window and positioned near the bottom.

15.2. Scrollbar User Interface

Types of Scrolling Motion

The default scrollbar is vertical, with *page buttons* at the top and bottom. To scroll, the user moves the cursor into the scrollbar (either the bar itself or one of the page buttons) and clicks one of the mouse buttons. The following table describes the available scrolling actions and how they are generated:

Table 15-1 *Scrolling Motions*

<i>Mouse Button</i>	<i>Cursor Location</i>	<i>Scrolling Action</i>
LEFT	page button	Line forward
RIGHT	page button	Line backward
MIDDLE	page button	Page forward
MIDDLE (shifted)	page button	Page backward
LEFT	bar	Line opposite cursor goes to top
RIGHT	bar	Top line comes to cursor
LEFT (shifted)	bar	Bottom line comes to cursor
RIGHT (shifted)	bar	Line opposite cursor goes to bottom
MIDDLE	bar	The line whose offset into the scrolling object approximates that of the cursor into the scrollbar is positioned at top ("thumbing").

Holding the button down within the scrollbar causes the cursor to change, previewing the scrolling action for that button. Releasing the button causes the scrolling action to be performed, or, if the user holds down the mouse button, the scrolling motion will start in repeating mode.

Undoing a Scroll

[Shift]-MIDDLE mouse button positions the viewing window to the most recent position which was left by an absolute motion (thumbing or undoing). The undoing position is initialized to the beginning of the scrollable object.

15.3. Creating, Destroying and Modifying Scrollbars

Scrollbars are created and destroyed with `scrollbar_create()` and `scrollbar_destroy()`. To take the simplest possible example, you get a default scrollbar (vertical, on the left edge of the subwindow, etc.) by calling:

```
Scrollbar bar;

bar = scrollbar_create(0);
```

You would destroy the scrollbar with the call:

```
scrollbar_destroy(bar);
```

The appearance and behavior of a given scrollbar is determined by the values of its attributes. Here's an example of a non-default scrollbar:

```
bar_1 = scrollbar_create(
    SCROLL_PLACEMENT,          SCROLL_EAST,
    SCROLL_BUBBLE_COLOR,       SCROLL_BLACK,
    SCROLL_BAR_DISPLAY_LEVEL,  SCROLL_ACTIVE,
    SCROLL_BUBBLE_DISPLAY_LEVEL, SCROLL_ACTIVE,
    SCROLL_DIRECTION,          SCROLL_VERTICAL,
    SCROLL_THICKNESS,          20,
    SCROLL_BUBBLE_MARGIN,      4,
    0),
```

In the above call, setting `SCROLL_PLACEMENT` to `SCROLL_EAST` will cause the scrollbar to appear on the right edge of the subwindow. The scrollbar will be 20 pixels wide with a black bubble 4 pixels from each edge of the bar. The bar and bubble will be shown only when the cursor is in the scrollbar.

You can modify and retrieve the attributes of a scrollbar with the two routines:

```
scrollbar_set(scrollbar, attributes)
    Scrollbar scrollbar;
    <attribute-list> attributes;

caddr_t
scrollbar_get(scrollbar, attribute)
    Scrollbar scrollbar;
    Scrollbar_attribute attributes;
```

If the scrollbar parameter is `NULL`, `scrollbar_get()` returns 0.

`SCROLL_RECT`, `SCROLL_THICKNESS`, `SCROLL_HEIGHT` and `SCROLL_WIDTH` do not have valid values until the scrollbar is passed into the subwindow. As a work-around for this problem, the special symbol `SCROLLBAR` has been provided. You can determine the default thickness of a scrollbar before it has been attached to a subwindow with the call:

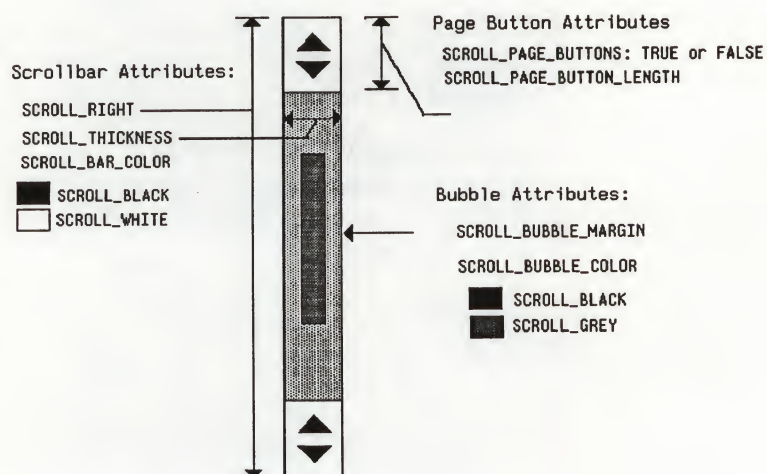
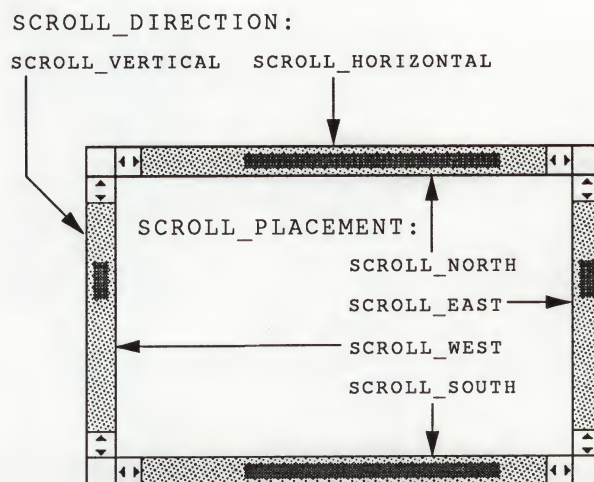
```
thickness = (int) scrollbar_get(SCROLLBAR, SCROLL_THICKNESS);
```

This convention is currently only implemented for `SCROLL_THICKNESS`.

If you set the `SCROLL_THICKNESS` attribute then you must also set the `SCROLL_DIRECTION` of the scrollbar, since the dimension of the scrollbar that is altered by `SCROLL_THICKNESS` depends on the orientation of the scrollbar.

The figures on the next page show some of the attributes controlling the visual appearance of a scrollbar.⁸⁹ Figure 15-2 illustrates the attributes that control the scrollbar appearance. Figure 15-3 illustrates the attributes that control the scrollbar placement.

⁸⁹ For a complete list of the scrollbar attributes see the *Scrollbar Attributes* table in Chapter 19, *SunView Interface Summary*.

Figure 15-2 *Attributes Controlling Scrollbar Appearance*Figure 15-3 *Scrollbar Placement Attributes*

15.4. Programmatic Scrolling

To scroll to a given location from your program, call:

```
scrollbar_scroll_to(scrollbar, new_view_start)
    Scrollbar scrollbar;
    long      new_view_start;
```

This routine saves the current value of `SCROLL_VIEW_START` as `SCROLL_LAST_VIEW_START`, sets `SCROLL_VIEW_START` to the value passed in as `new_view_start`, and posts a scroll event to the scrollbar's client (i.e. the canvas, panel or text subwindow) using the Notifier. This has the same effect as if the user had requested a scroll to `new_view_start`.

The Selection Service

The Selection Service	279
16.1. Getting the Primary Selection	280
16.2. Setting the Primary Selection	280



The Selection Service

The Selection Service provides for flexible communication among window applications. You can use the Selection Service to query and manipulate the selections the user has made.

This chapter gives only the simplest example of using the Selection Service. To find out more about the Selection Service and the other functionality it provides, refer to Chapter 9 of the *SunView 1 System Programmer's Guide*.

The definitions necessary to use the Selection Service are found in the include file `<suntool/seln.h>`.

16.1. Getting the Primary Selection

The primary selection is the selection made by the user without holding down any of the function keys, and is indicated with reverse-video highlighting on the screen.

The routine below is taken from the program *filer*, listed in Appendix A. It retrieves the primary selection by first asking the Selection Service which window has the primary selection, then asking that window for the characters that are in the selection, saving them in a static buffer, and returning a pointer to that buffer:

```
#define <suntool/seln.h>

#define MAX_FILENAME_LEN 256

char *
get_selection()
{
    static char    filename[MAX_FILENAME_LEN];
    Seln_holder    holder;
    Seln_request    *buffer;

    holder = seln_inquire(SELN_PRIMARY);
    buffer = seln_ask(&holder, SELN_REQ_CONTENTS_ASCII, 0, 0);

    strncpy(filename,
            buffer->data + sizeof(Seln_attribute),
            MAX_FILENAME_LEN);

    return (filename);
}
```

This example has been kept simple by removing error checking. The code relies on the fact that if there is no primary selection, or the Selection Service process is not running, or the holder of the primary selection failed to return the selection string, then the buffer returned by `seln_ask()` will have an empty string for the selection characters.

The routine also assumes that the selection will be no more than 256 characters long. `seln_ask()` will handle selections of up to about 2000 characters. To find out how to handle arbitrarily large selections, or selections other than the primary selection, refer to the *SunView 1 System Programmer's Guide*.

16.2. Setting the Primary Selection

For an example of a program which sets, and responds to queries about, the selection, see *seln_demo*, in Chapter 9 of the *SunView 1 System Programmer's Guide*.

The Notifier

The Notifier	283
Header Files	283
Related Documentation	283
17.1. When to Use the Notifier	285
17.2. Restrictions	285
Don't Call.	285
Don't Catch.	286
17.3. Overview	287
How the Notifier Works	287
Client Handles	287
Types of Interaction	287
17.4. Event Handling	288
Child Process Control Events	288
"Reaping" Dead Processes	288
Results from a Process	289
Input-Pending Events (pipes)	290
Example: Reading a Pipe	290
Closing the Pipe	291
Signal Events	291
A <code>signal()</code> Replacement for Notifier Compatibility	291
Example: Writing to a Pipe	292
Asynchronous Event Handling	293
Timeout Events	294

Example: Periodic Feedback	294
Polling	295
Checking the Interval Timer	296
Turning the Interval Timer Off	296
17.5. Interposition	296
How Interposition Works	296
Monitoring a Frame's State	297
Example: Interposing on Open/Close	297
Discarding the Default Action	299
Interposing on Resize Events	299
Example: <i>resize_demo</i>	299
Modifying a Frame's Destruction	299
Destroy Events	300
Checking	300
Destruction	300
A Typical Destroy Handler	300
Example: Interposing a Client Destroy Handler	301
17.6. Porting Programs to SunView	303
Explicit Dispatching	303
Implicit Dispatching	303
Getting Out	304
17.7. Error Handling	305
Error Codes	305
Handling Errors	305
Debugging	306
NOTIFY_ERROR_ABORT	306
Stop in <code>notify_perror()</code> or <code>fprintf(3S)</code>	306
<code>notify_dump</code>	306

The Notifier

The Notifier is a general-purpose mechanism for distributing events to a collection of clients within a process. It detects events in which its clients have expressed an interest, and dispatches these events to the proper clients, queuing client processing so that clients respond to events in a predictable order.

An overview of the notification-based model is given in Chapter 2, *The SunView Model*.

To encourage the porting of existing applications, the Notifier has provisions to allow programs to run in the Notifier environment without inverting their control structure. See Section 17.6, *Porting Programs to SunView*.

Header Files

The definitions for the Notifier are contained in the file `<sunwindow/notify.h>`, which will be included indirectly when you include `<suntool/sunview.h>`.⁹⁰

Related Documentation

This chapter will suffice for the majority of SunView applications. See the chapters titled *Advanced Notifier Usage* and *The Agent and Tiles* in the *SunView 1 System Programmer's Guide* for more information on the Notifier and SunView's usage of it. When looking up Notifier-related information, look first in the index to this book, then in the index to the *SunView 1 System Programmer's Guide*.

Summary Listing and Table

To give you a feeling for what you can do with the Notifier, the following page contains a list of the available Notifier functions. Many of these are discussed in the rest of this chapter and elsewhere (use the *Index* to check). All are briefly described with their arguments in the *Notifier Functions* table beginning on page 343 in Chapter 19, *SunView Interface Summary*.

⁹⁰ For those programmers utilizing the Notifier outside of SunView (a perfectly reasonable thing to do), the code that implements the Notifier is found in `/usr/lib/libsunwindow.a`.

Notifier Functions

```
notify_default_wait3(client, pid, status, rusage)
notify_dispatch()
notify_do_dispatch()
notify_interpose_destroy_func(client, destroy_func)
notify_interpose_event_func(client, event_func, type)
notify_itimer_value(client, which, value)
notify_next_destroy_func(client, status)
notify_no_dispatch()
notify_perror(s)
notify_set_destroy_func(client, destroy_func)
notify_set_exception_func(client, exception_func, fd)
notify_set_input_func(client, input_func, fd)
notify_set_itimer_func(client, itimer_func, which, value, ovalue)
notify_set_signal_func(client, signal_func, signal, when)
notify_start()
notify_stop()
notify_set_output_func(client, output_func, fd)
notify_set_wait3_func(client, wait3_func, pid)
notify_veto_destroy(client)
```

17.1. When to Use the Notifier

Since the Notifier is used by the SunView libraries, any program that uses SunView implicitly uses the Notifier. You will have to use the Notifier explicitly if you want to do any of the following:

- Catch signals, e.g., SIGCONT.
- Notice state changes in processes that your process has spawned, e.g., a child process has died.
- Read and write through file descriptors, e.g., using pipes.
- Receive notification of the expiration of an interval timer, e.g., so that you can provide some blinking user feedback.
- Extend, modify or monitor SunView Notifier clients, e.g., noticing when a frame is opened, closed or about to be destroyed.
- Use a non-notification-based control structure while running under SunView, e.g., porting programs to SunView.

17.2. Restrictions

The Notifier imposes some restrictions on its clients which designers should be aware of when developing software to work in the Notifier environment. These restrictions exist so that the application and the Notifier don't interfere with each other. More precisely, since the Notifier is multiplexing access to user process resources, the application needs to respect this effort so as not to violate the sharing mechanism.

Don't Call...

Assuming an environment with multiple clients with an unknown notifier usage pattern, you should not use any of the following system calls or C library routines:⁹¹

- | | |
|--------------|--|
| signal(3) | The Notifier is catching signals on the behalf of its clients. If you set up your own signal handler over the one that the Notifier has set up then the Notifier will never notice the signal. |
| sigvec(2) | The same applies for sigvec(2) as does for signal(3), above. |
| setitimer(2) | The Notifier is managing two of the process's interval timers on the behalf of its many clients. If you access an interval timer directly, the Notifier could miss a timeout. Use <code>notify_set_itimer_func()</code> instead of <code>setitimer(2)</code> . |
| alarm(3) | Because <code>alarm(3)</code> sets the process's interval timer directly, the same applies for <code>alarm(3)</code> as does for <code>setitimer(2)</code> , above. |
| getitimer(2) | When using a notifier-managed interval timer, you should call <code>notify_itimer_value()</code> to get its current status. Otherwise, you can get inaccurate results. |
| wait3(2) | The Notifier notices child process state changes on behalf of its clients. If you do your own <code>wait3(2)</code> , then the notifier may never notice the change in a child |

⁹¹ A future release may provide modified versions of some of these forbidden routines that will allow their use without restriction. However, the restrictions described in *Don't Catch...*, below, will continue to be germane. A `signal()` Replacement for Notifier Compatibility, in Section 17.4, provides a code patch for programs that catch signals.

process or you may get a change of state for a child process in which you have no interest. Use `notify_set_wait3_func()` instead of `wait3(2)`.

`wait(2)` The same applies for `wait(2)` as does for `wait3(2)`, above.

`ioctl(2)(..., FIONBIO, ...)` This call sets the blocking status of a file descriptor. The Notifier needs to know the blocking status of a file descriptor in order to determine if there is activity on it. `fcntl(2)` has an analogous request that should be used instead of `ioctl(2)`.

`ioctl(2)(..., FIOASYNC, ...)` This call controls a file descriptor's asynchronous io mode setting. The Notifier needs to know this mode in order to determine if there is activity on it. `fcntl(2)` has an analogous request that should be used instead of `ioctl(2)`.

`system(3)` In the SunOS, this function calls `signal(3)` and `wait(2)`. Hence you should avoid using this for the reasons mentioned above. Calls to `system(3)` should be replaced with something like the following.

```
if((pid = vfork()) == 0) {
    (void) execl("/bin/sh", "sh", "-c", str, (char *)0);
    _exit(127);
}
notify_set_wait3_func(me, my_handler, pid);
```

Don't Catch...

Clients should not have to catch any of the following signals. If you are, then you are probably also making one of the forbidden calls described above. You might also be utilizing the Notifier inappropriately if you think that you have to catch any of these signals. The Notifier catches these signals itself under a variety of circumstances:

SIGALRM Caught by the Notifier's interval timer manager. Use `notify_set_itimer_func()` instead.

SIGVTALRM The same applies for **SIGVTALRM** as does for **SIGALRM** above.

SIGTERM Caught by the Notifier so that it can tell its clients that the process is going away. Use `notify_set_destroy_func()` if that is why you are catching **SIGTERM**.

SIGCHLD Caught by the Notifier so that it can do child process management. Use `notify_set_wait3_func()` instead.

SIGIO Caught by the Notifier so that it can manage its file descriptors that are running in asynchronous io mode. Use `notify_set_input_func()`⁹² or `notify_set_output_func()` if you want to know when there is activity on your file descriptor.

SIGURG Caught by the Notifier so that it can dispatch exception activity on a file descriptor to its clients. Use `notify_set_exception_func()` if you are looking for out-of-band communications when using a socket.

⁹² Do not use a NULL client handle when you use `notify_set_input_func()` or the Notifier will go into an infinite loop.

If you think you have to catch one of these signals, then be sure to use `notify_set_signal_func()`.

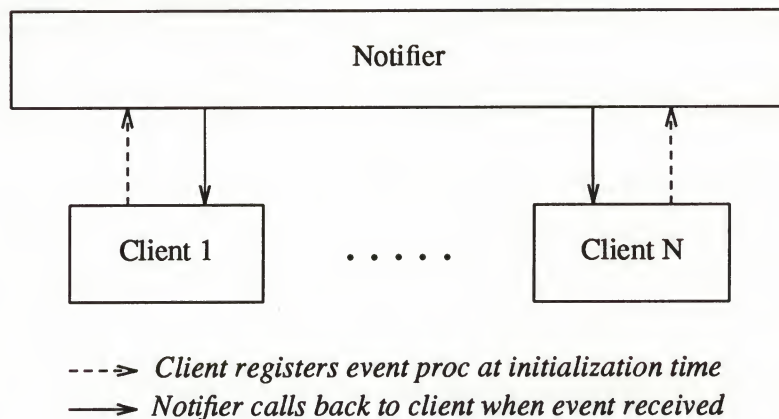
17.3. Overview

How the Notifier Works

Before it can receive events, a client must advise the Notifier of the types of events in which it is interested. It does this by registering an *event handler* function (which it must supply) for each type of event in which it is interested. When an event occurs, the Notifier calls the event handler appropriate to the type of event.

Figure 17-1 shows an overview of how the notification mechanism works.

Figure 17-1 Overview of Notification



Client Handles

The Notifier uses a *client handle* as the unique identifier for a given client. The Notifier, without interpreting the client handle in any way, uses it to associate each event with the event handler for a given client.

The only requirement for a client handle is that it must be unique (within a process). Since a program text address or the address of an allocated data block are guaranteed to be unique, they can be used. Since stack addresses are not in general guaranteed to be unique they should not be used. Internally, SunView uses the object handles returned from `window_create()` as notifier client handles.

Types of Interaction

Client interaction with the Notifier falls into the following functional areas:

- Event handling — A client may receive events and respond to them via *event handlers*. Event handlers do the bulk of the work in the Notifier environment. The various types of events are in Section 17.4, *Event Handling*.
- Interposition — A client may request that the Notifier install a special type of event handler (supplied by the client) to be inserted (or *interposed*) ahead of the current event handler for a given type of event and client. This allows clients to screen incoming events and redirect them, and to monitor and change the status of other clients. Examples of interposition may be found below under *Monitoring a Frame's State*.

- Notifier control — A client may exercise control over when dispatching of events occurs. See Section 17.6, *Porting Programs to SunView*.

17.4. Event Handling

This section describes how to be notified of UNIX-related events and notifier supported destroy events (see Chapter 6, *Handling Input*, for a description of SunView-defined events). UNIX events are low-level occurrences that are meaningful at the level of the operating system. These include signals (software interrupts), input pending on a file descriptor, output completed on a file descriptor, tasks associated with managing child processes, and tasks associated with managing interval timers.

A client establishes an interest in a certain type of event by registering an event handler procedure to respond to it. The event handler for a given type of event has a mandatory calling sequence, as described below. All event handlers return a value of either `NOTIFY_DONE` or `NOTIFY_IGNORED` depending on whether the event was acted on in some way or failed to provoke any action, respectively.

When registering an event handler, the registration procedure returns a pointer to the function that was in place previous to the current call. On initialization, the Notifier sets up its internal tables by registering “dummy” functions as placeholders. These dummy functions are no-op functions with no harmful side-effects. The first time a client registers a given type of event handler, it will receive a pointer to a “dummy” function.

The following sections describe common usages of various types of events.

Child Process Control Events

Let's say that you want to fork a process to perform some processing on your behalf. UNIX requires that you perform some housekeeping of that process. The minimum housekeeping required is to notice when that process dies and “reap” it. You can register a *wait3 event* handler,⁹³ which the Notifier will call whenever a child process changes state (e.g. dies), by calling:

```
Notify_func
notify_set_wait3_func(client, wait3_func, pid)
    static Notify_client client;
    Notify_func          wait3_func;
    int                  pid;
```

“Reaping” Dead Processes

Clients using child process control which simply need to perform the required reaping after a child process dies can use the predefined `notify_default_wait3()` as their *wait3 event* handler. For example:

⁹³ The name *wait3 event* originates from the `wait3(2)` system call.

```

#include <sunwindow/notify.h>

static int my_client_object;
static Notify_client *me = &my_client_object;

int pid;

if ((pid = my_fork()))
    (void) notify_set_wait3_func(me, notify_default_wait3,
                                pid);

/* Start dispatching events */
(void) notify_start();

```

This is sufficient to have your child process reaped on its death. The Notifier automatically removes a dead process's wait3 event handler from its internal data structures.

NOTE *The use of me as a client handle is arbitrary, but illustrates one method of generating a unique client handle.*

Results from a Process

A more interesting application might actually receive some results from the process it forked. In this case, the application would supply its own wait3 event handler⁹⁴. For example:

```

#include <sunwindow/notify.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
static Notify_value my_wait3_handler();

/* Register a wait3 event handler */
(void) notify_set_wait3_func(me, my_wait3_handler, pid);
/* Start dispatching events */
(void) notify_start();

static Notify_value
my_wait3_handler(me, pid, status, rusage)
    Notify_client me;
    int pid;
    union wait *status;
    struct rusage *rusage;

{
    if (WIFEXITED(*status)) {
        /* Child process exited with return code */
        my_return_code_handler(me, status->w_retcode);
        /* Tell the notifier that you handled this event */
        return (NOTIFY_DONE);
    }
    /* Tell the notifier that you ignored this event */
    return (NOTIFY_IGNORED);
}

```

⁹⁴ See the wait(2) manual page for details of union wait and struct rusage.

Input-Pending Events (pipes)

A program may need to know when there is input pending on a file descriptor — for instance, on one end of a pipe. Let's extend our previous example a bit to include reading data from a pipe connected to a process that we have forked. You can register an input-pending event handler which the Notifier will call whenever there is input pending on a file descriptor⁹⁵ by calling:

```
Notify_func
notify_set_input_func(client, input_func, fd)
    Notify_client client;
    Notify_func input_func;
    int fd;
```

The calling sequence for the `input_func()` you supply is as follows:

```
Notify_value
input_func(client, fd)
    Notify_client client;
    int fd;
```

Example: Reading a Pipe

```
#include <sunwindow/notify.h>

static Notify_value my_pipe_reader();

    int fildes[2];
    /* Create a pipe */
    if (pipe(fildes) == -1) {
        perror("pipe");
        exit(1);
    }
    /* Register an input-pending event handler */
    (void) notify_set_input_func(me, my_pipe_reader, fildes[0]);
    ... do fork and dispatching from wait3 event example ...
    ... do fork and dispatching from wait3 event example ...

static Notify_value
my_pipe_reader(me, fd)
    Notify_client me;
    int fd;
    /* Read the pipe (fd) */
    ...
    /* Tell the notifier that the input event is handled */
    return (NOTIFY_DONE);
```

In the above example, the application uses the Notifier to read from the pipe because it doesn't want to block on input pending on the pipe. In the case of a SunView program, the program wants to return back to the Notifier's central dispatching loop so that the user can interact with the window while waiting for input to become available on the pipe.

⁹⁵ The file descriptor can be in blocking or non-blocking mode, or in asynchronous mode; the Notifier handles both as long as you have used `fcntl(2)` to set the modes.

Closing the Pipe

When you close any file descriptor that has been registered with the Notifier you should *unregister* it. To do this, call `notify_set_input_func()` with a `notify_func` of `NOTIFY_FUNC_NULL`.⁹⁶

Signal Events

Signals are UNIX software interrupts. The Notifier multiplexes access to the UNIX signal mechanism. A client may ask to be notified that a UNIX signal occurred either when it is received (asynchronously) and/or later during normal processing (synchronously).

Clients may define and register a signal event handler to respond to any UNIX signal desired. However, many of the signals that you might catch in a traditional UNIX program may be being caught for you by the Notifier (see *Don't catch* above).

CAUTION

Clients of the Notifier must not directly catch any UNIX signals using `signal(3)` or `sigvec(2)`. Regardless of whether clients choose synchronous or asynchronous signal notification, they must use the signal event mechanism described in this section. See Section 17.2, *Restrictions*.

You can register a signal event handler which the Notifier will call whenever a signal has been caught by calling:

```
Notify_func
notify_set_signal_func(client, signal_func, signal, when)
    Notify_client      client;
    Notify_func        signal_func;
    int                signal;
    Notify_signal_mode when;
```

`when` can be either `NOTIFY_SYNC` or `NOTIFY_ASYNC`. `NOTIFY_SYNC` causes notification during normal processing, that is, the delivering of the signal is delayed, so that your program doesn't receive it at an arbitrary time. `NOTIFY_ASYNC` causes notification immediately as the signal is received, — this mode mimics the UNIX `signal(3)` semantics.

A `signal()` Replacement for Notifier Compatibility

You should rewrite applications to use `notify_set_signal_func()`. However, the Notifier routine `notify_set_signal_func()` does not fully emulate the `signal(3)` function. It does not handle errors the same way `signal(3)` does. Errors from `signal(3)` are indicated by a `-1` return value, and the value of `errno` is set to `EINVAL`.

The errors for `notify_set_signal_func()` are not communicated back to the caller, but error messages are printed. For example, if the signal number is not valid, the Notifier prints

```
Bad signal number
```

but its return value indicates success; the `signal(3)` system call does not print a message, but returns `-1` and sets `errno` to `EINVAL`. As another example, if

⁹⁶ This method of passing in a `NOTIFY_FUNC_NULL` to unregister an event handler from the Notifier works for any type of event.

SIGKILL or SIGSTOP are ignored or a handler supplied, the Notifier prints

Notifier assertion botched: Unexpected error: sigvec

but its return value indicates success, while `signal(3)` does not print a message, returns value of `-1`, and sets `errno` to `EINVAL`.

The work-around is to use the following replacement function for the C library version of `signal(3)`. This code converts `signal()` calls into `notify_set_signal_func()` calls. Explicitly loading this code will override the loading of the C library's version of `signal()`. This approach works only if all the signal handlers registered by `signal()` only look at the first argument passed to them when a signal is received. Also, no Notifier client handle may be a small integer.

```
#include <sunwindow/notify.h>
#include <errno.h>

int (*
signal(sig, func)) ()
int sig, (*func) ();

    if ( (sig < 1 || sig > NSIG) ||
        (sig == SIGKILL || sig == SIGSTOP) ) {
        errno = EINVAL;
        return (BADSIG);
    }
    if (sig == SIGCONT && func == SIG_IGN) {
        errno = EINVAL;
        return (BADSIG);
    }
    return ((int (*)())notify_set_signal_func(sig, func,
                                              sig, NOTIFY_ASYNC));
```

Example: Writing to a Pipe

Let's extend our on-going example by writing on the pipe. Writing to a pipe that has no process at the other end to receive the message causes a SIGPIPE to be generated by UNIX. By default, an uncaught SIGPIPE causes a premature process termination. So, we are going to catch SIGPIPE so that our process doesn't get killed if we start a process that dies.⁹⁷

⁹⁷ We are glossing over the part about actually writing to the pipe. If we wanted to write something to the pipe and then get some notification about when the write had actually completed (i.e., the other process had read it) we would use the `notify_set_output_func()` call. The calling sequences for this routine and its event handler are exactly the same as those for `notify_set_input_func()` (previously described).

```

#include <sunwindow/notify.h>
#include <signal.h>

static Notify_value my_sigpipe_handler();

    ... do pipe from input-pending example ...
    ... do notify_set_input_func from input-pending example ...
    ... do fork from wait3 event example ...

/* Register a signal event handler */
(void) notify_set_signal_func(me, my_sigpipe_handler,
                             SIGPIPE, NOTIFY_ASYNC);

/* Write a message on the pipe */

...

/* Start dispatching events */
(void) notify_start();

static Notify_value
my_sigpipe_handler(me, signal, when)
    Notify_client    me;
    int              signal;
    Notify_signal_mode when;

/*
 * This is a no-op function meant only to prevent us from
 * being killed because we didn't have a SIGPIPE handler.
 */
return (NOTIFY_IGNORED);

```

This example wouldn't actually show `my_sigpipe_handler()` being called unless you set up the child process to die right away.

Asynchronous Event Handling

An asynchronous signal notification can come at any time (unless blocked using `sigblock(2)`). This means that the client can be executing code at any arbitrary place. Great care must be exercised during asynchronous processing.

It is rarely safe to do much of anything in response to an asynchronous signal. Unless your program has taken steps to protect its data from asynchronous access, the only safe thing to do is to set a flag indicating that the signal has been received.

When in an asynchronous signal event handler, the signal context and signal code is available from the follow routines:

```

int
notify_get_signal_code()

struct sigcontext *
notify_get_signal_context()

```

The return values of these routines are undefined if called from a synchronous signal event handler.

Timeout Events

A client may require notification of an expired timer based on real time (approximate elapsed wall clock time; `ITIMER_REAL`) or on process virtual time (CPU time used by this process; `ITIMER_VIRTUAL`). To receive this type of notification, the client must define and register a timeout event handler.

```
Notify_func
notify_set_itimer_func(client, itimer_func, which, value,
                        ovalue)
Notify_client      client;
Notify_func        itimer_func;
int                which;
struct itimerval *value, *ovalue;
```

The semantics of `which`, `value` and `ovalue` parallel the arguments to `setitimer(2)` (see the `getitimer(2)` manual page). `which` is either `ITIMER_REAL` or `ITIMER_VIRTUAL`.

Example: Periodic Feedback

As an example, we want to provide some form of blinking feedback. We do this by setting up an interval timer when we want to blink. We turn the internal timer off when we no longer need the blinking.⁹⁸

⁹⁸ This code segment should be wrapped in, say, a panel notify procedure, in order to be actually run.

```

#include <sunwindow/notify.h>
#include <sys/time.h>

static int blinking_required; /* blinking desired */
static int blinking;          /* blinking enabled */
#define ITIMER_NULL ((struct itimerval *)0)
static Notify_value my_blinker();

    if (blinking_required && !blinking) {

        struct itimerval blink_timer;

        /* Set up interval with which to RELOAD the timer */
        blink_timer.it_interval.tv_usec = 0;
        blink_timer.it_interval.tv_sec = 1;

        /* Set up INITIAL value with which to SET the timer */
        blink_timer.it_value.tv_usec = 0;
        blink_timer.it_value.tv_sec = 1;

        /* Turn on interval timer for client */
        (void) notify_set_itimer_func(me, my_blinker,
            ITIMER_REAL, &blink_timer, ITIMER_NULL);
        blinking = 1;

    } else if (!blinking_required && blinking) {

        /* Turn off interval timer for client */
        (void) notify_set_itimer_func(me, my_blinker,
            ITIMER_REAL, ITIMER_NULL, ITIMER_NULL);
        blinking = 0;

    }
static Notify_value
my_blinker(me, which)
    Notify_client me;
    int which;
    /* Do the blink */
    ...
    return (NOTIFY_DONE);

```

Polling

Interval timers can be used to set up a polling situation. There is a special value argument to `notify_set_itimer_func()` that tells the Notifier to call you as often and as quickly as possible. This value is the address of the following constant:

```
struct itimerval NOTIFY_POLLING_ITIMER; /*{{0,1},{0,1}}*/
```

This high speed polling can consume all of your machine's available CPU time, but may be appropriate for high speed animation. It is used in the program *spheres*, which shows one way to convert an old SunWindows gfx subwindow-based program to SunView. *spheres* is explained in Appendix C, *Converting SunWindows Programs to SunView*, and is listed in full in Appendix A, *Example Programs*.

Checking the Interval Timer

The following function checks on the state of an interval timer by returning its current state in the structure pointed to by value.

```
Notify_error  
notify_itimer_value(client, which, value)  
    Notify_client    client;  
    int              which;  
    struct itimerval *value;
```

Turning the Interval Timer Off

If you specify an interval timer with its `it_interval` structure set to `{0, 0}`, the Notifier flushes any knowledge of the interval timer after it delivers the timeout notification. Otherwise, supplying a NULL interval timer pointer to `notify_set_itimer_func()` will turn the timer off.

17.5. Interposition

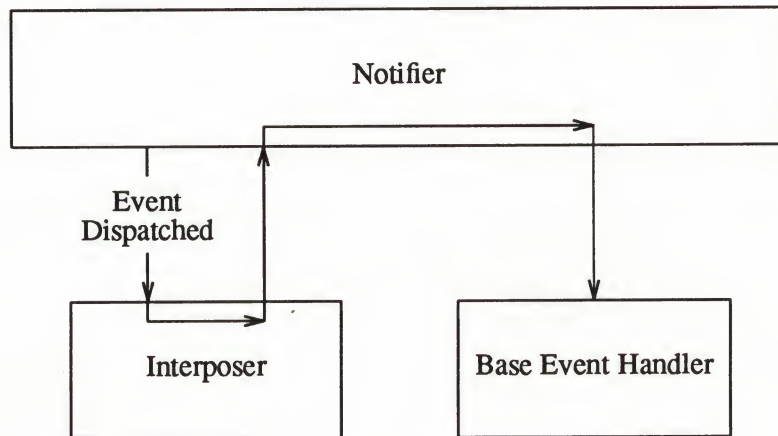
SunView window objects utilize the Notifier for much of their communication and cooperation. The Notifier provides a mechanism called *interposition*, with which you can intercept control of the internal communications within SunView. Interposition is a powerful way to both monitor and modify window behavior in ways that extend the functionality of a window object.

Interposition allows a client to intercept an event before it reaches the *base event handler*. The base event handler is the one set originally by a client. The client can call the base event handler before or after its own handling of the event, or not at all. Clients may use interposition to monitor and filter events coming in to an event handler and/or to modify a series of actions based on the results of some calculation.

How Interposition Works

A client requests that the Notifier install an interposer function, supplied by the client, for a specified client and type of event. When an event arrives, the Notifier calls the function at the top of the wait list for that client and that type of event. An interposed routine may (indirectly) call the next function in the interposition sequence and receive its results.

Figure 17-2 illustrates the flow of control with interposition. Note that the interposer could have stopped the flow of control to the base event handler.

Figure 17-2 *Flow of Control in Interposition*

Monitoring a Frame's State

You can notice when a frame opens or closes by interposing in front of the frame's *client event* handler. The client event handler is a SunView specific event handler which is built on top of the Notifier's general client event mechanism.⁹⁹ To install an interposer call the following routine:

```

Notify_error
notify_interpose_event_func(client, event_func, type)
    Notify_client      client;
    Notify_func        event_func;
    Notify_event_type  type;
  
```

`client` must be the handle of the Notifier client in front of which you are interposing. In SunView, this is the handle returned from `window_create()`.¹⁰⁰ `type` is always `NOTIFY_SAFE` for SunView clients.

Example: Interposing on Open/Close

Let's say that the application is displaying some animation, and wants to do the necessary computation only when the frame is open. It can use interposition to notice when the frame opens or closes.

The program *spheres* (which shows one way to convert an old SunWindows gfx subwindow-based program to SunView) uses this technique to stop shading an image when its frame is closed. It is explained in Appendix C, *Converting SunWindows Programs to SunView*, and is listed in full in Appendix A, *Example Programs*.

Another example appears on the following page. Note the the call to `notify_next_event_func()`, which transfers control to the frame's client event handler through the Notifier. `notify_next_event_func()` takes the same arguments as the interposer.

⁹⁹ The stream of events sent to a client event handler is described in Chapter 6, *Handling Input*.

¹⁰⁰ It could also be the handle returned from the call to `scrollbar_create()`.


```

#include <suntool/sunview.h>

static Notify_value my_frame_interposer();

...
Frame frame;

/* Create the frame */
frame = window_create(0, FRAME,
...,
0);

...
/* Interpose in front of the frame's event handler */
(void) notify_interpose_event_func(frame,
my_frame_interposer, NOTIFY_SAFE);
/* Show frame and start dispatching events */
window_main_loop(frame);

static Notify_value
my_frame_interposer(frame, event, arg, type)
Frame frame;
Event *event;
Notify_arg arg;
Notify_event_type type;

int closed_initial, closed_current;
Notify_value value;
{
/* Determine initial state of frame */
closed_initial = (int) window_get(frame, FRAME_CLOSED);

/* Let frame operate on the event */
value = notify_next_event_func(frame, event, arg, type);

/* Determine current state of frame */
closed_current = (int) window_get(frame, FRAME_CLOSED);

/* Change animation if states differ */
if (closed_initial != closed_current) {
if (closed_current) {
/* Turn off animation because closed */
(void) notify_set_itimer_func(me, my_animation,
ITIMER_REAL, ITIMER_NULL, ITIMER_NULL);
} else {
/* Turn on animation because opened */
(void) notify_set_itimer_func(me, my_animation,
ITIMER_REAL, &NOTIFY_POLLING_ITIMER,
ITIMER_NULL);
}
}
return (value);
}

```

Discarding the Default Action

In the example on the preceding page, you wanted the base event handler to handle the event (so that the frame gets closed/opened). If the interposed function replaces the base event handler, and you don't want the base event handler to be called at all, your interposed procedure should not call `notify_next_event()`. For example, your interposed function might handle scroll events itself, so you would not want the base event handler to perform an additional scroll.

Interposing on Resize Events

Another common use of interposition is to give your application more control over the layout of its subwindows. The code is very similar. You call `notify_interpose_event_func()` to interpose your event handler. In the event handler, the following fragment could be used:

```
value = notify_next_event_func(frame, event, arg, type);
if (event_action(event) == WIN_RESIZE)
    resize(frame);
return(value);
```

Let the default event handler handle the event, then check if the event is a resize event. If so, call your own `resize()` procedure to lay out the subwindows.

NOTE A `WIN_RESIZE` event is not generated until the frame is resized. If you want your resize procedure to be called when the window first appears you must do so yourself. This is different from a canvas with the `CANVAS_RESIZE` attribute set, whose resize procedure is called the first time the canvas is displayed.

If the user manually adjusts subwindow sizes using **Control**-middle mouse button, no `WIN_RESIZE` event is generated. You can disallow subwindow resizing by setting the `FRAME_SUBWINDOWS_ADJUSTABLE` attribute to `FALSE`.

Example: *resize_demo*

The program *resize_demo* shows how to achieve more complex window layouts than possible using window layout attributes. It is listed in Appendix A, *Example Programs*.

Modifying a Frame's Destruction

Suppose an application must detect when the user selects the 'Quit' menu item in the frame menu, in order to perform some application-specific confirmation. We have to interpose in front of the frame's *client destroy event* handler using the following routine.

```
Notify_error
notify_interpose_destroy_func(client, destroy_func)
    Notify_client client;
    Notify_func destroy_func;
```

First, however, you need to understand client destroy events.

Destroy Events

The Notifier can tell each client to destroy itself. It is possible for a destroy event handler to receive two calls concerning client destruction: one call may be a status inquiry and the other a demand for termination. Destroy event handlers use a status code to determine whether the caller demands actual termination (`DESTROY_CLEANUP` or `DESTROY_PROCESS_DEATH`), or simply requires an indication if it is feasible for the client to terminate at present (`DESTROY_CHECKING`).

Checking

If the status argument indicates an inquiry and the client cannot terminate at present, the destroy event handler should call `notify_veto_destroy()`, indicating that termination would not be advisable at this time, and return normally. If the status argument indicates an inquiry and the client can terminate at present, then the destroy handler should do nothing; a subsequent call will tell the client to actually destroy itself.

This veto option is used, for example, to give a text subwindow the chance to ask the user to confirm the saving of any editing changes when quitting a tool.

Destruction

If the status argument is not `DESTROY_CHECKING` then the client is being told to destroy itself. If status is `DESTROY_PROCESS_DEATH` then the client can count on the entire process dying and so should do whatever it needs to do to cleanup its outside entanglements, e.g., update a file used by other processes. Since the entire process is dying, one might choose to not release all the resources used within the process, e.g., dynamically allocated memory. However, if status is `DESTROY_CLEANUP` then the client is being asked to destroy itself and be very tidy about cleaning up all the process internal resources that it is using, as well as its outside entanglements.

A Typical Destroy Handler

A typical destroy handler looks like the following:

```
Notify_value
common_destroy_func(client, status)
    Notify_client client;
    Destroy_status status;
    if (status == DESTROY_CHECKING) {
        if (/* Don't want to go away now */)
            notify_veto_destroy(client);
    } else {
        /* Always release external commitments */
        if (status == DESTROY_CLEANUP)
            /* Conditionally release internal resources */
    }
    return (NOTIFY_DONE);
```

Example: Interposing a Client Destroy Handler

Now we can present the example of interposing in front of the frame's client destroy event handler. In addition to doing our own confirmation, we prevent double confirmation by suppressing the frame's default confirmation.

Note that after having the destroy OK'd by the user, we call `notify_next_destroy_func()` before returning. This allows other subwindows to request confirmation.

The code appears on the following page.


```

#include <suntool/sunview.h>

static Notify_value my_frame_destroyer();

...
/*
 * Interpose in front of the frame's destroy event handler
 */
(void) notify_interpose_destroy_func(frame,
                                     my_frame_destroyer);
/* Show frame and start dispatching events */
window_main_loop(frame);

...

static Notify_value
my_frame_destroyer(frame, status)
    Frame      frame;
    Destroy_status status;

{
    if (status == DESTROY_CHECKING) {
        if (my internal state requires confirmation) {
            /*
             * Request confirmation from the user
             * (see window_loop() in the index).
             */

            ...

            if (destroy OK'd by user) {
                /* Tell frame not to do confirmation */
                window_set(frame, FRAME_NO_CONFIRM, TRUE, 0);
            } else {
                /*
                 * Tell the Notifier that the destroy has
                 * been vetoed.
                 */
                (void) notify_veto_destroy(frame);
                /*
                 * Return now so that the destroy event
                 * never reaches the frame's destroy handler.
                 */
                return (NOTIFY_DONE);
            }
        } else {
            /* Let frame do normal confirmation */
            window_set(frame, FRAME_NO_CONFIRM, FALSE, 0);
        }
    }
    /* Let frame get destroy event */
    return (notify_next_destroy_func(frame, status));
}

```

17.6. Porting Programs to SunView

Most programs that are ported to SunView are not notification-based. They are traditional programs that maintain strict control over the inner control loop. Much of the state of such programs is preserved on the stack in the form of local variables. The Notifier supports this form of programming so that you can use SunView packages without inverting the control structure of your program to be notification-based.

Explicit Dispatching

The simplest way to convert a program to coexist with the Notifier is called *explicit dispatching*. This approach replaces the call to `window_main_loop()`, which usually doesn't return until the application terminates, with the following bit of code:

```
#include <suntool/sunview.h>

static int my_done;

extern Notify_error notify_dispatch();

/* Make the frame visible on the screen */
window_set(frame, WIN_SHOW, TRUE, 0);
while (!my_done) {
    ...
    /* Dispatch events managed by the Notifier */
    (void) notify_dispatch();
    ...
}
```

`notify_dispatch()` goes once around the Notifier's internal loop, dispatches any pending events, and returns. You should try to have `notify_dispatch()` called at least once every 1/4 second so that good interactive response with SunView windows can be maintained.

The program *bounce* (which shows one way to convert an old SunWindows gfx subwindow-based program to SunView) uses explicit dispatching. It is explained in Appendix C, *Converting SunWindows Programs to SunView*, and is given in full in in Appendix A, *Example Programs*.

Implicit Dispatching

Explicit dispatching is good when you are performing some computationally intensive processing and you want to occasionally give the user a chance to interact with your program. There is another method of interacting with the Notifier that is useful when you simply want the Notifier to take care of its clients and block until there is something of interest to you. This is called *implicit dispatching*.

This time, we replace the call to `window_main_loop()` with the following bit of code:


```
#include <suntool/sunview.h>

static int my_done;

window_set(frame, WIN_SHOW, TRUE, 0);
/* Enable implicit dispatching */
(void) notify_do_dispatch();
while (!my_done) {
    char c;
    ...
    /* read allows implicit dispatching by Notifier */
    if ((n = read(0/*stdin*/, &c, 1)) < 0)
        perror("my_program");
    ...
}
```

`notify_do_dispatch()` allows the Notifier to dispatch events from within the calls to `read(2)` or `select(2)`. The Notifier's versions of `read(2)` and `select(2)` won't return until the normal versions would. They can block exactly like the normal versions.

`notify_no_dispatch()` (it takes no arguments) prevents the Notifier from dispatching events from within the call to `read(2)` or `select(2)`.

Getting Out

When you use either of these dispatching approaches, you will need to find out when the frame is 'Quit' by the user, in order to know when to terminate your program. To do so, interpose in front of the frame's destroy event handler, as in the previous section, so that you can notice when the frame goes away. At this point you can call `notify_stop()` to break the `read(2)` or `select(2)` out of a blocking state.

```

#include <suntool/sunview.h>

static    int my_done;

static Notify_value my_notice_destroy();

    /*
    * Interpose in front of the frame's destroy event handler
    */
    (void) notify_interpose_destroy_func(frame,
                                         my_notice_destroy);

static Notify_value
my_notice_destroy(frame, status)
    Frame          frame;
    Destroy_status status;

{
    if (status != DESTROY_CHECKING) {
        /* Set my flag so that I terminate my loop soon */
        my_done = 1;
        /* Stop the notifier if blocked on read or select */
        (void) notify_stop();
    }
    /* Let frame get destroy event */
    return (notify_next_destroy_func(frame, status));
}

```

17.7. Error Handling

Error Codes

Every call to a notifier routine returns a value that indicates success or failure. Routines that return an enumerated type called `Notify_error` deliver `NOTIFY_OK` (zero) to indicate a successful operation, while any other value indicates failure. Routines that return function pointers deliver a non-null value to indicate success, while a value of `NOTIFY_FUNC_NULL` indicates an error condition.

When an error occurs, the global variable `notify_errno` describes the failure. The Notifier sets `notify_errno` much like UNIX system calls set the global `errno`; that is, the Notifier only sets `notify_errno` when it detects an error and does not reset it to `NOTIFY_OK` on a successful operation. A table in the *SunView 1 System Programmer's Guide* lists each possible value of `notify_errno` and its meaning.

Handling Errors

Most of the errors returned from the Notifier indicate a programmer error, e.g., the arguments are not valid. Often the best approach for the client is to print a message if the return value is non-zero and exit. The procedure `notify_perror()` takes a string which is printed to `stderr`, followed by a colon, followed by a terse description of `notify_errno`. This is done in a manner analogous to the UNIX `perror(3)` call.

Debugging

Here are some debugging hints that may prove useful when programming:

`NOTIFY_ERROR_ABORT`

Setting the environment variable `NOTIFY_ERROR_ABORT` to YES will cause the Notifier to abort with a core dump when the Notifier detects an error. This is useful if there is some race condition that produces notifier error messages that you are having a hard time tracking down.

Stop in `notify_perror()`
or `fprintf(3S)`

If you are getting notifier error messages, but don't know from where, try putting a break point on the entry to either `notify_perror()` or `fprintf(3S)`. Trace the stack to see what provoked the message.

`notify_dump`

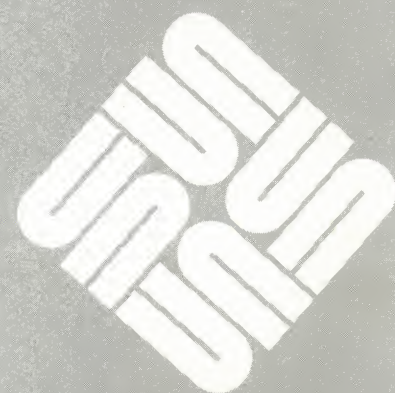
The following call can be made from the debugger or your program to dump a printout of the state of the Notifier:

```
void
notify_dump(client, type, file)
    Notify_client client;
    int             type;
    FILE            *file;
```

The state of `client` is dumped to `file` based on the value of `type`. If `client` is 0 then all clients are dumped. If `type` is 1 then all the registered event handlers are dumped. If `type` is 2 then all the events pending for delivery are dumped. If `type` is 3 then both the registered event handlers and the events pending for delivery are dumped. If `file` is 1 then `stdout` is assumed. If `file` is 2 then `stderr` is assumed. To be able to call `notify_dump()` you need to reference it from some place in your program so that it gets loaded into your binary.

Attribute Utilities

Attribute Utilities	309
18.1. Character Unit Macros	309
18.2. Creating Reusable Attribute Lists	310
Default Attributes	311
18.3. Maximum Attribute List Size	311



Attribute Utilities

This chapter describes macros and functions that are provided as utilities to be used with attributes.

18.1. Character Unit Macros

By default in SunView, coordinate specification attributes interpret their values in pixel units. For applications that don't make heavy use of images, it is usually more convenient to specify positions in character units — columns and rows rather than xs and ys. To this end two macros `ATTR_ROW()` and `ATTR_COL()` are provided, which interpret their arguments as rows or columns, respectively, and convert the value to the corresponding number of pixels, based on the subwindow's font, as specified by `WIN_FONT`. `ATTR_ROW()` and `ATTR_COL()` take as arguments any expression yielding an integer. The use of these macros as an operand in an expression is restricted to adding a pixel offset (e.g., `ATTR_ROW(5) + 2`). Examples of legal and illegal usage are given in the table below.

Table 18-1 *Example uses of the `ATTR_ROW()` and `ATTR_COL()` macros*

<i>Attribute/Value</i>	<i>Interpretation</i>
<code>PANEL_ITEM_X, 5</code>	5 pixels from left
<code>PANEL_ITEM_Y, 10</code>	10 pixels from top
<code>PANEL_ITEM_X, ATTR_COL(5)</code>	column 5
<code>PANEL_ITEM_X, ATTR_COL(-5)</code>	column -5
<code>PANEL_ITEM_X, ATTR_COL(5+2)</code>	column 7
<code>PANEL_ITEM_X, ATTR_COL(5)+2</code>	2 pixels to right of col 5
<code>PANEL_ITEM_X, ATTR_COL(5)-1</code>	1 pixel to left of col 5
<code>PANEL_ITEM_Y, ATTR_ROW(10)</code>	row 10
<code>PANEL_ITEM_Y, ATTR_ROW(-10)</code>	row -10
<code>PANEL_ITEM_Y, ATTR_ROW(10+2)</code>	row 12
<code>PANEL_ITEM_Y, ATTR_ROW(10)+2</code>	2 pixels down from row 10
<code>PANEL_ITEM_Y, ATTR_ROW(10)-1</code>	1 pixel up from row 10
<code>PANEL_ITEM_X, ATTR_COL(10)+ATTR_COL(2)</code>	<i>illegal</i>
<code>PANEL_ITEM_X, 2*ATTR_COL(10)</code>	<i>illegal</i>

NOTE `ATTR_ROW()` and `ATTR_COL()` treat their arguments as character positions rather than lengths. In other words, when you use `ATTR_ROW(5)`, the pixel value that is computed includes the top margin. Similarly, the pixel value computed using `ATTR_COL(5)` includes the left margin.

These macros can be used with the panel attributes or the window attributes such as WIN_X, WIN_HEIGHT, etc.

Both the attributes and the ATTR_ROW() and ATTR_COL() macros are zero-based — that is, the first row is row zero.

If you want to use lengths rather than positions, you can use the alternate macros ATTR_ROWS() and ATTR_COLS(). Examples of the differences between the character position and length macros are given in the table below.

Table 18-2 *Example uses of the ATTR_ROWS() and ATTR_COLS() macros*

<i>Attribute/Value</i>	<i>Interpretation</i>
WIN_WIDTH, ATTR_COL(80)	80 characters wide + left margin
WIN_WIDTH, ATTR_COLS(80)	exactly 80 characters wide
WIN_HEIGHT, ATTR_ROW(24)	24 lines high + top margin
WIN_HEIGHT, ATTR_ROWS(24)	exactly 24 lines high
PANEL_ITEM_X, ATTR_COL(5)	col 5 (left margin + 5 character widths)
PANEL_ITEM_X, ATTR_COLS(5)	5 character widths from the left edge
PANEL_ITEM_Y, ATTR_ROW(5)	row 5 (top margin + 5 row heights)
PANEL_ITEM_Y, ATTR_ROWS(5)	5 row heights from the top edge

18.2. Creating Reusable Attribute Lists

You may want to create an attribute list that can be passed to different routines. You can do this either by creating the list explicitly, or by using the routine attr_create_list().

To create an attribute list explicitly, define a static array of char *, which is initialized (or later filled in with) the desired attribute/value pairs. Note that non-string values must be coerced to type char *:

```
static char *attributes[] = {
    (char*)PANEL_LABEL_STRING, "Name: ",
    (char*)PANEL_VALUE,        "Goofy ",
    (char*)PANEL_NOTIFY_PROC,  (char *)name_item_proc,
    0 }
```

To make an attribute list dynamically, use:

```
Attr_avlist
attr_create_list(attributes)
    <attribute-list> attributes;
```

attr_create_list() allocates storage for the list it returns. It is up to you to free this storage when no longer needed, as in:

```

Attr_avlist list;

list = attr_create_list(PANEL_LABEL_BOLD, TRUE, 0);
...
...
free(list);

```

The `free()` procedure is the standard UNIX `free(3)` routine.

Default Attributes

The code below shows how to use `attr_create_list()` in conjunction with the attribute `ATTR_LIST` to support default attributes in a panel.

```

int          text_proc(), name_proc();
Panel_item   name_item, address_item;
Pixfont      *big_font, *small_font;
Attr_avlist  defaults;

defaults = attr_create_list(
                                PANEL_SHOW_ITEM,    FALSE,
                                PANEL_LABEL_FONT,    big_font,
                                PANEL_VALUE_FONT,    small_font,
                                PANEL_NOTIFY_PROC,    text_proc,
                                0);

name_item = panel_create_item(PANEL_TEXT,
                                ATTR_LIST,           defaults,
                                PANEL_NOTIFY_PROC,    name_proc,
                                0);

address_item = panel_create_item(PANEL_TEXT,
                                ATTR_LIST,           defaults,
                                PANEL_SHOW_ITEM,      TRUE,
                                PANEL_VALUE_FONT,     big_font,
                                0);

```

The special attribute `ATTR_LIST` takes as its value an attribute list. In the above example, first an attribute list called `defaults` is created. Then, by mentioning `defaults` first in the attribute lists for subsequent item creation calls, each item takes on those default attributes. Subsequent references to an attribute override the setting in `defaults` since the last value mentioned for an attribute is the one which takes effect.

18.3. Maximum Attribute List Size

The maximum length of attribute-value lists supported by the SunView packages (see `ATTR_STANDARD_SIZE` in `<sunwindow/attr.h>`) is 250. If the number of attributes in a list you pass to SunView exceeds this size, the attribute package prints

```

Number of attributes(nnn) in the attr list exceeds
the maximum number(nnn) specified. Exit!

```

on standard output and exits with exit status 1.

SunView Interface Summary

SunView Interface Summary	315
Alert Tables	316
Attributes	316
Functions	318
Canvas Tables	319
Attributes	319
Functions and Macros	320
Cursor Tables	321
Attributes	321
Functions	323
Data Types	324
Icon Tables	328
Attributes	328
Functions and Macros	329
Input Event Tables	330
Event Codes	330
Event Descriptors	333
Input-Related Window Attributes	334
Menu Tables	335
Attributes	335
Item Attributes	339
Functions	341
Notifier Functions Table	343



Notifier Functions Table	343
Panel Tables	346
Attributes	346
Generic Panel Item Attributes	347
Choice and Toggle Item Attributes	349
Slider Item Attributes	351
Text Item Attributes	352
Functions and Macros	353
Pixwin Tables	356
Pixwin Drawing Functions and Macros Table	356
Pixwin Color Manipulation Functions Table	360
Attributes	362
Functions and Macros	365
Text Subwindow Tables	366
Attributes	366
Textsw_action Attributes	370
Textsw_status Values	371
Functions	372
TTY Subwindow Tables	376
Attributes	376
Functions	376
Special Escape Sequences	377
Window Tables	379
Attributes	379
Frame Attributes	382
Functions and Macros	384
Command Line Frame Arguments	386

SunView Interface Summary

This chapter contains tables summarizing the data types, functions and attributes which comprise the SunView programmatic interface.¹⁰¹

The tables correspond to the chapters in this book, but are in *alphabetical* order: Alerts, Canvases, Cursors, Data Types, Icons, Input (including events and input-related window attributes), Menus, the Notifier, Panels, Pixwins, Scrollbars, Text Subwindows, TTY Subwindows and Windows (including frames and frame command line arguments).

Note that the order of the chapters is different than the order of the tables. The chapter on windows (including frames) comes first, followed by canvases, input, pixwins, text subwindows, panels, alerts, tty subwindows, menus, cursors, icons, scrollbars, the Selection Service, and the Notifier.

Within each topic, the attribute tables come first, then the functions and macros, then miscellaneous tables.

To help distinguish where one table ends and another begins, the start of each table is marked with a horizontal grey bar.

¹⁰¹ This chapter does not include a table for the Selection Service functions; see the *SunView System Programmer's Guide* for a complete discussion of the Selection Service interface.

Table 19-1 Alert Attributes

Attribute	Type	Description
ALERT_BUTTON	char *, int	A string to be displayed in a button and a value to associate with it. The value specified with the string will be returned when the button is selected. The value may be any integer, but should not be a value predefined by the alerts package; that is, not ALERT_YES, ALERT_NO, ALERT_FAILED, or ALERT_DEFAULT_TRIGGERED). See the values given in the <i>Alert Functions</i> table.
ALERT_BUTTON_FONT	Pixfont *	Font used for buttons. Default is the font specified for menus, which is <i>MenuFont</i> in defaultsedit or screen.b.14 if no default is specified.
ALERT_BUTTON_NO	char *	A string that is associated with the accelerated NO (cancel, don't do it) button which is triggered via a keyboard accelerator. The value returned if this button is selected (or the accelerator is triggered) will be ALERT_NO. Only one instance of this attribute is allowed.
ALERT_BUTTON_YES	char *	A string to associate with the accelerated YES (ie. confirm, continue, do it) button which is also triggered via a keyboard accelerator. The value returned when this button is selected (or the accelerator is triggered) will be ALERT_YES. Only one instance of this attribute is allowed.
ALERT_MESSAGE_FONT	Pixfont *	Font used for message strings. The default is the same as Client Frame (if specified) otherwise it is the same as <i>SunViewFont</i> .
ALERT_MESSAGE_STRINGS	list char*	Strings to be displayed in the message area of the alert panel. The default is to be determined.
ALERT_MESSAGE_STRINGS_ARRAY_PTR	array char*	Same as ALERT_MESSAGE_STRINGS except the client need not know the actual strings being passed, just that the value is pointer to first of null terminated array of strings. The alerts package will cast the value into a type char **.
ALERT_NO_BEEPING	int	Allows the client to specify that no beeping should take place regardless of defaults database setting. The default for this option is FALSE; that is, beep however many times database specifies.

Table 19-1 *Alert Attributes—Continued*

<i>Attribute</i>	<i>Type</i>	<i>Description</i>
ALERT_OPTIONAL	boolean	Specifies whether an optional alert will be enabled or disabled. You make an alert a courtesy alert by specifying the ALERT_OPTIONAL attribute in the attribute list passed to alert_prompt().
ALERT_POSITION	int	<p>Specifies the position of the alert. Default is ALERT_CLIENT_CENTERED unless client_frame = NULL. NULL causes the alert to default to ALERT_SCREEN_CENTERED regardless of this setting.</p> <p>Possible values that may be passed are: ALERT_SCREEN_CENTERED, ALERT_CLIENT_CENTERED, and ALERT_CLIENT_OFFSET. Use WIN_X and WIN_Y for the offset attributes. This position describes where the “center” of an alert should be.</p>
ALERT_TRIGGER	int	This special attribute allows the client to specify a SunView event which should cause the alert to return. The default is not to return unless an actual button has been selected or the other YES/NO accelerators are seen. When this event is triggered, the value returned will be ALERT_TRIGGERED.

Table 19-2 *Alert Functions*

Definition	Description
<pre> int alert_prompt(client_frame, event, attributes) Frame client_frame; Event *event; <attribute-list> attributes; </pre>	<p>Displays alert and doesn't return until the user pushes a button, or its trigger or the default has been seen. A value of ALERT_FAILED is returned if alert_prompt() failed for any reason, otherwise equivalent to ordinal value of button which caused return (ie. button actually selected, or default button if default action triggered return). The client_frame may be NULL (see ALERT_POSITION for consequences). The event will be completely filled in at the time the alert_prompt() returns.</p> <p>The possible status values that may be returned from this function are: the (int) value passed with every ALERT_BUTTON attribute; ALERT_YES, if a confirm button or trigger was pushed; ALERT_NO, if a cancel button or trigger was pushed; ALERT_FAILED, if the alert failed to pop up; and ALERT_TRIGGERED, if a keyboard accelerator was used.</p>

Table 19-3 *Canvas Attributes*

<i>Attribute</i>	<i>Type</i>	<i>Description</i>
CANVAS_AUTO_CLEAR	boolean	If TRUE, repaint area of canvas pixwin is cleared before. repaint proc is called. Default: TRUE unless the canvas is retained.
CANVAS_AUTO_EXPAND	boolean	If TRUE, canvas width and height are never allowed to be less than the edges of the canvas pixwin. Default: TRUE.
CANVAS_AUTO_SHRINK	boolean	If TRUE, canvas width and height are never allowed to be greater than the edges of the canvas pixwin. Default: TRUE.
CANVAS_FAST_MONO	boolean	If TRUE, tells canvases and graphics subwindows to use the monochrome overlay plane of the Sun-3/110 display. Default: FALSE.
CANVAS_FIXED_IMAGE	boolean	If TRUE, canvas package assumes that client is drawing a fixed-size image whose rendering does not depend on the size of the canvas. Default: TRUE.
CANVAS_HEIGHT	int	Height of object being drawn. Default: height of usable window, which is $WIN_HEIGHT - (SCROLL_THICKNESS \text{ of } WIN_HORIZONTAL_SCROLLBAR) - CANVAS_MARGIN * 2$.
CANVAS_MARGIN	int	Margin to leave around the canvas pixwin from inside of window. Default: 0.
CANVAS_PIXWIN	Pixwin *	Pixwin for drawing. Get only.
CANVAS_REPAINT_PROC	(procedure)	Called when repaint needed, even if retained. Default: NULL. Form: <pre> repaint_proc(canvas, pixwin, repaint_area) Canvas canvas; Pixwin *pixwin; Rectlist *repaint_area;</pre>
CANVAS_RESIZE_PROC	(procedure)	Called when canvas width or height changes. Default: NULL. Form: <pre> resize_proc(canvas, width, height) Canvas canvas; int width; int height;</pre>
CANVAS_RETAINED	boolean	If TRUE, image is backed up for repaint. Default: TRUE.
CANVAS_WIDTH	int	Width of object being drawn. Default: width of usable window, which is $WIN_WIDTH - (SCROLL_THICKNESS \text{ of } WIN_VERTICAL_SCROLLBAR) - CANVAS_MARGIN * 2$.

Table 19-4 *Canvas Functions and Macros*

<i>Definition</i>	<i>Description</i>
<pre> Event * canvas_event(canvas, event) Canvas canvas; Event *event; </pre>	<p>Translates the coordinates of event from the space of the canvas subwindow to the space of the logical canvas (which may be larger and scrollable). That is, the client passes in a pointer to an event, then the function does an event_set_x(event, translated_x) and an event_set_y(event, translated_y). It then returns the same pointer that was passed in.</p>
<pre> Pixwin * canvas_pixwin(canvas) Canvas canvas; </pre>	<p>Returns the pixwin to use when drawing into the canvas with the pw_*() routines.</p>
<pre> Event * canvas_window_event(canvas, event) Canvas canvas; Event *event; </pre>	<p>Translates the coordinates of event to the space of the canvas subwindow from the space of the logical canvas.</p>

Table 19-5 *Cursor Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
CURSOR_CROSSHAIR_BORDER_GRAVITY	boolean	Crosshairs stick to borders. Default: FALSE.
CURSOR_CROSSHAIR_COLOR	int	Color for crosshairs. Default: 1. (Note: the color displayed depends on the settings in your colormap segment).
CURSOR_CROSSHAIR_GAP	int	Half-length of space to leave untouched from intersection of crosshairs. Value of CURSOR_TO_EDGE extends crosshairs to edge of cursor rect. Default: 0.
CURSOR_CROSSHAIR_LENGTH	int	Half-length of crosshairs. Default: CURSOR_TO_EDGE.
CURSOR_CROSSHAIR_OP	int	Raster op for drawing crosshairs. Default: PIX_SRC.
CURSOR_CROSSHAIR_THICKNESS	int	Thickness of crosshairs. Maximum value is CURSOR_MAX_HAIR_THICKNESS (5). Default: 1.
CURSOR_FULLSCREEN	boolean	Clip crosshairs to edge of screen not window. Default: FALSE.
CURSOR_HORIZ_HAIR_BORDER_GRAVITY	boolean	Horizontal crosshair sticks to borders. Default: FALSE.
CURSOR_HORIZ_HAIR_COLOR	int	See CURSOR_HORIZ_HAIR_COLOR
CURSOR_HORIZ_HAIR_GAP	int	See CURSOR_CROSSHAIR_GAP.
CURSOR_HORIZ_HAIR_LENGTH	int	See CURSOR_CROSSHAIR_LENGTH.
CURSOR_HORIZ_HAIR_OP	int	Raster op for drawing horizontal crosshair. Default: PIX_SRC.
CURSOR_HORIZ_HAIR_THICKNESS	int	See CURSOR_CROSSHAIR_THICKNESS.
CURSOR_IMAGE	Pixrect *	Cursor's image. Default: 16 x 16 x 1 blank pixrect.
CURSOR_OP	int	Raster op for drawing cursor image. Default: PIX_SRC PIX_DST.
CURSOR_SHOW_CROSSHAIRS	boolean	Show or don't show crosshairs. Default: FALSE.
CURSOR_SHOW_CURSOR	boolean	Show or don't show cursor image. Default: TRUE.
CURSOR_SHOW_HORIZ_HAIR	boolean	Show or don't show horizontal crosshair. Default: FALSE.
CURSOR_SHOW_VERT_HAIR	boolean	Show or don't show vertical crosshair. Default: FALSE.

Table 19-5 *Cursor Attributes—Continued*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
CURSOR_VERT_HAIR_BORDER_GRAVITY	boolean	Vertical crosshair sticks to borders. Default: FALSE.
CURSOR_VERT_HAIR_COLOR	int	See CURSOR_CROSSHAIR_COLOR
CURSOR_VERT_HAIR_GAP	int	See CURSOR_CROSSHAIR_GAP.
CURSOR_VERT_HAIR_LENGTH	int	See CURSOR_CROSSHAIR_LENGTH.
CURSOR_VERT_HAIR_OP	int	Raster op for drawing vertical crosshair. Default: PIX_SRC.
CURSOR_VERT_HAIR_THICKNESS	int	See CURSOR_CROSSHAIR_THICKNESS.
CURSOR_XHOT	int	Hot spot x coordinate. Default: 0.
CURSOR_YHOT	int	Hot spot y coordinate. Default: 0.

Table 19-6 *Cursor Functions*

<i>Definition</i>	<i>Description</i>
<pre>Cursor cursor_copy(src_cursor) Cursor src_cursor;</pre>	Creates and returns a copy of <code>src_cursor</code> .
<pre>Cursor cursor_create(attributes) <attribute-list> attributes;</pre>	Creates and returns the opaque handle to a cursor.
<pre>void cursor_destroy(cursor) Cursor cursor;</pre>	Destroys <code>cursor</code> .
<pre>caddr_t cursor_get(cursor, attribute) Cursor cursor; Cursor_attribute attribute;</pre>	Retrieves the value for an attribute of <code>cursor</code> .
<pre>void cursor_set(cursor, attributes) Cursor cursor; <attribute-list> attributes;</pre>	Sets the value for one or more attributes of <code>cursor</code> . <code>attributes</code> is a null-terminated attribute list.

Table 19-7 *Data Types*

<i>Data Type</i>	<i>Description</i>
Canvas	Pointer to an opaque structure which describes a canvas.
Cursor	Pointer to an opaque structure which describes a cursor.
Destroy_status	Enumeration: DESTROY_PROCESS_DEATH, DESTROY_CHECKING, or DESTROY_CLEANUP.
Event	The structure which describes an input event: <pre>typedef struct inputevent { short ie_code; short ie_flags; short ie_shiftmask; short ie_lock; short ie_locy; struct timeval ie_time; } Event;</pre>
Frame	Pointer to an opaque structure which describes a frame.
Icon	Pointer to an opaque structure which describes a icon.
Inputmask	Mask specifying which input events a window will receive.
Menu	Pointer to an opaque structure which describes a menu.
Menu_attribute	One of the menu attributes (MENU_*).
Menu_generate	Enumerated type of the operation parameter passed to generate procs: MENU_CREATE, MENU_DESTROY, MENU_NOTIFY_CREATE or MENU_NOTIFY_DESTROY.
Menu_item	Pointer to an opaque structure which describes a menu item.
Notify_arg	Opaque client optional argument.
Notify_destroy	Enumeration: NOTIFY_SAFE, NOTIFY_IMMEDIATE. (See also Notify_event_type).
Notify_event	Opaque client event.

Table 19-7 *Data Types—Continued*

<i>Data Type</i>	<i>Description</i>
Notify_error	Enumeration of errors for notifier functions: NOTIFY_OK, NOTIFY_UNKNOWN_CLIENT, NOTIFY_NO_CONDITION, NOTIFY_BAD_ITIMER, NOTIFY_BAD_SIGNAL, NOTIFY_NOT_STARTED, NOTIFY_DESTROY_VETOED, NOTIFY_INTERNAL_ERROR, NOTIFY_SRCH, NOTIFY_BADF, NOTIFY_NOMEM, NOTIFY_INVAL, or NOTIFY_FUNC_LIMIT.
Notify_event_type	Enumeration: NOTIFY_SAFE, NOTIFY_IMMEDIATE.
Notify_func	Notifier function.
Notify_signal_mode	Enumeration: NOTIFY_SYNC, NOTIFY_ASYNC.
Notify_value	Enumeration of possible return values for client notify procs: NOTIFY_DONE, NOTIFY_IGNORED, or NOTIFY_UNEXPECTED.
Panel	Pointer to an opaque structure which describes a panel.
Panel_attribute	One of the panel attributes (PANEL_*).
Panel_item	Pointer to an opaque structure which describes a panel item.
Panel_setting	Enumerated type returned by panel_text_notify(); also type of repaint argument to panel_paint(). See the <i>Panels</i> chapter and <suntool/panel.h>.
Pixfont	The structure representing a font (for definition see the Pixrect Reference Manual).
Pixrect	The basic object of pixel manipulation in the SunView window system. Pixrects include both a rectangular array of pixels and the means of accessing operations for manipulating those pixels (for definition see the Pixrect Reference Manual).
Pixwin	The basic imaging element of the SunView window system. While, for historical reasons, its fields are public, clients should treat it as an opaque handle.
Rect	The structure describing a rectangle: <pre>typedef struct rect { short r_left; short r_top; short r_width; short r_height; } Rect;</pre>

Table 19-7 Data Types—Continued

<i>Data Type</i>	<i>Description</i>
Rectlist	<p>A list of rectangles:</p> <pre>typedef struct rectlist { short rl_x, rl_y; Rectnode *rl_head; Rectnode *rl_tail; Rect rl_bound; } Rectlist; typedef struct rectnode { Rectnode *rn_next; Rect rn_rect; } Rectnode;</pre>
Scroll_motion	<p>Enumerated type representing possible scrolling motions:</p> <p>SCROLL_ABSOLUTE, SCROLL_FORWARD, SCROLL_MAX_TO_POINT, SCROLL_PAGE_FORWARD, SCROLL_LINE_FORWARD, SCROLL_BACKWARD, SCROLL_POINT_TO_MAX, SCROLL_PAGE_BACKWARD, or SCROLL_LINE_BACKWARD.</p>
Scrollbar	The opaque handle for a scrollbar.
Scrollbar_attribute	One of the scrollbar attributes (SCROLL_*).
Scrollbar_setting	The value of an enumerated type scrollbar attribute.
Textsw	Pointer to an opaque structure which describes a text subwindow.
Textsw_index	An index for a character within a text subwindow.
Textsw_enum	<p>Enumerated type for various text subwindow attribute values:</p> <p>TEXTSW_ALWAYS, TEXTSW_NEVER, TEXTSW_ONLY, TEXTSW_IF_AUTO_SCROLL, TEXTSW_CLIP, TEXTSW_WRAP_AT_CHAR, TEXTSW_WRAP_AT_WORD.</p>
Textsw_status	<p>Enumeration describing the status of text subwindow operations:</p> <p>TEXTSW_STATUS_OKAY, TEXTSW_STATUS_BAD_ATTR, TEXTSW_STATUS_BAD_ATTR_VALUE, TEXTSW_STATUS_CANNOT_ALLOCATE, TEXTSW_STATUS_CANNOT_OPEN_INPUT, or TEXTSW_STATUS_OTHER_ERROR,</p>
Tty	Pointer to an opaque structure which describes a tty subwindow.
Window	Pointer to an opaque structure which describes a window.

Table 19-7 *Data Types— Continued*

<i>Data Type</i>	<i>Description</i>
Window_attribute	One of the window attributes (WIN_*).
Window_type	Type of window, retrieved via the WIN_TYPE attribute. One of: FRAME_TYPE, PANEL_TYPE, CANVAS_TYPE, TEXTSW_TYPE, or TTY_TYPE.

Table 19-8 *Icon Attributes*

<i>Attribute</i>	<i>Type</i>	<i>Description</i>
ICON_FONT	Pixfont *	Font for icon's label.
ICON_HEIGHT	int	Icon's height in pixels. Default: 64.
ICON_IMAGE	Pixrect *	Memory pixrect for icon's image.
ICON_IMAGE_RECT	Rect *	Rect for icon's image. Default: origin (0, 0), width 64, height 64.
ICON_LABEL	char *	Icon's label.
ICON_LABEL_RECT	Rect *	Rect for icon's label. Default: origin (0, 0), width 0, height 0.
ICON_WIDTH	int	Icon's width in pixels. Default: 64.

Table 19-9 *Icon Functions and Macros*

<i>Definition</i>	<i>Description</i>
<pre>Icon icon_create(attributes) <attribute-list> attributes;</pre>	Creates and returns the opaque handle to an icon.
<pre>int icon_destroy(icon) Icon icon;</pre>	Destroys icon.
<pre>caddr_t icon_get(icon, attribute) Icon icon; Icon_attribute attribute;</pre>	Retrieves the value for an attribute of icon.
<pre>int icon_set(icon, attributes) Icon icon; <attribute-list> attributes;</pre>	Sets the value for one or more attributes of icon. attributes is a null-terminated attribute list.
<pre>extern static struct mpr_data DEFINE_ICON_FROM_IMAGE(name, image) static short icon_image[];</pre>	Macro that creates a static memory pixrect icon from image; the latter typically is generated by including a file created by iconedit. Note: you must pass the <i>address</i> of icon to the icon routines, since the Icon object is a pointer.

Table 19-10 Event Codes

<i>Event Code</i>	<i>Description</i>	<i>Value (for debugging)</i>
ASCII_FIRST	Marks beginning of ASCII range	0
ASCII_LAST	Marks end of ASCII range	127
META_FIRST	Marks beginning of META range	128
META_LAST	Marks end of META range	255
ACTION_ERASE_CHAR_BACKWARD	Erase char to the left of caret	31744
ACTION_ERASE_CHAR_FORWARD	Erase char to the right of caret	31745
ACTION_ERASE_WORD_BACKWARD	Erase word to the left of caret	31746
ACTION_ERASE_WORD_FORWARD	Erase word to the right of caret	31747
ACTION_ERASE_LINE_BACKWARD	Erase to the beginning of the line	31748
ACTION_ERASE_LINE_END	Erase to the end of the line	31749
ACTION_GO_CHAR_BACKWARD	Move the caret one character to the left	31752
ACTION_GO_CHAR_FORWARD	Move the caret one character to the right	31753
ACTION_GO_WORD_BACKWARD	Move the caret one word to the left	31754
ACTION_GO_WORD_END	Move the caret to the end of the word	31756
ACTION_GO_WORD_FORWARD	Move the caret one word to the right	31755
ACTION_GO_LINE_BACKWARD	Move the caret to the start of the line	31757
ACTION_GO_LINE_END	Move the caret to the end of the line	31759
ACTION_GO_LINE_FORWARD	Move the caret to the start of the next line	31758
ACTION_GO_COLUMN_BACKWARD	Move the caret up one line, maintaining column position	31761
ACTION_GO_COLUMN_FORWARD	Move the caret down one line, maintaining column position	31762
ACTION_GO_DOCUMENT_START	Move the caret to the beginning of the text	31763
ACTION_GO_DOCUMENT_END	Move the caret to the end of the text	31764
ACTION_STOP	Stop the operation	31767
ACTION_AGAIN	Repeat previous operation	31768
ACTION_PROPS	Show property sheet window	31769
ACTION_UNDO	Undo previous operation	31770
ACTION_FRONT	Bring window to the front of the desktop	31772
ACTION_BACK	Put the window at the back of the desktop	31773
ACTION_OPEN	Open a window from its icon form or close if already open)	31775
ACTION_CLOSE	Close a window to an icon	31776
ACTION_COPY	Copy the selection to the clipboard	31774
ACTION_PASTE	Copy clipboard contents to the insertion point	31777
ACTION_CUT	Delete the selection, put on clipboard	31781
ACTION_COPY_THEN_PASTE	Copies then pastes text	31784
ACTION_FIND_FORWARD	Find the text selection to the right of the caret	31779
ACTION_FIND_BACKWARD	Find the text selection to the left of the caret	31778
ACTION_FIND_AND_REPLACE	Show find and replace window	31780
ACTION_SELECT_FIELD_FORWARD	Select the next delimited field	31783

Table 19-10 *Event Codes—Continued*

<i>Event Code</i>	<i>Description</i>	<i>Value (for debugging)</i>
ACTION_SELECT_FIELD_BACKWARD	Select the previous delimited field	31782
ACTION_MATCH_DELIMITER	Selects text up to a matching delimiter	31894
ACTION_QUOTE	Causes next event in the input stream to pass untranslated by the keymapping system	31898
ACTION_EMPTY	Causes the subwindow to be emptied	31899
ACTION_STORE	Stores the specified selection as a new file	31785
ACTION_LOAD	Loads the specified selection as a new file	31786
ACTION_GET_FILENAME	Gets the selected filename	31788
ACTION_SET_DIRECTORY	Sets the directory to the selection	31788
ACTION_INCLUDE_FILE	Selects the current line (in pending-delete mode) and attempts to insert the file described by that selection	31891
ACTION_CAPS_LOCK	Toggle caps lock state	31895
PANEL_EVENT_CANCEL	The panel or panel item is no longer “current”	32000
PANEL_EVENT_MOVE_IN	The panel or panel item was entered with no mouse buttons down	32001
PANEL_EVENT_DRAG_IN	The panel or panel item was entered with one or more mouse buttons down	32002
SCROLL_REQUEST	Scrolling has been requested	32256
SCROLL_ENTER	Locator (mouse) has moved into the scrollbar	32257
SCROLL_EXIT	Locator (mouse) has moved out of the scrollbar	32258
LOC_MOVE	Locator (mouse) has moved	32512
LOC_STILL	Locator (mouse) has been still for 1/5 second	32513
LOC_WINENTER	Locator (mouse) has entered window	32514
LOC_WINEXIT	Locator (mouse) has exited window	32515
LOC_DRAG	Locator (mouse) has moved while a button was down	32516
LOC_RGNENTER	Locator (mouse) has entered a region of the window	32519
LOC_RGNEXIT	Locator (mouse) has exited a region of the window	32520
LOC_TRAJECTORY	Inhibits the collapse of mouse motions; clients receive LOC_TRAJECTORY events for every locator motion the window system detects.	32523
WIN_REPAINT	Some portion of window requires repainting	32517
WIN_RESIZE	Window has been resized	32518
WIN_STOP	User has pressed the <i>stop</i> key	32522
KBD_REQUEST	Window is about to become the focus of keyboard input	32526
KBD_USE	Window is now the focus of keyboard input	32524
KBD_DONE	Window is no longer the focus of keyboard input	32525
SHIFT_LEFT	Left shift key changed state	32530
SHIFT_RIGHT	Right shift key changed state	32531
SHIFT_CTRL	Control key changed state	32532
SHIFT_META	Meta key changed state	32534
SHIFT_LOCK	Shift lock key changed state	32529

Table 19-10 *Event Codes—Continued*

<i>Event Code</i>	<i>Description</i>	<i>Value (for debugging)</i>
SHIFT_CAPSLOCK	Caps lock key changed state	32528
BUT(i)	Locator (mouse) buttons 1–10	BUT(1) is 32544
MS_LEFT	Left mouse button	32544
MS_MIDDLE	Middle mouse button	32545
MS_RIGHT	Right mouse button	32546
KEY_LEFT(i)	Left function keys 1–15	KEY_LEFT(1) is 32554
KEY_RIGHT(i)	Right function keys 1–15	KEY_RIGHT(1) is 32570
KEY_TOP(i)	Top function keys 1–15	KEY_TOP(1) is 32586

Table 19-11 *Event Descriptors*

<i>Event Descriptor</i>	<i>Explanation</i>
WIN_NO_EVENTS	Clears input mask — no events will be accepted. Note: the effect is the same whether used with a <i>consume</i> or an <i>ignore</i> attribute. A new window has a cleared input mask.
WIN_ASCII_EVENTS	All ASCII events. ASCII events that occur while the META key is depressed are reported with codes in the META range. In addition, cursor control keys and function keys are reported as ANSI escape sequences: a sequence of events whose codes are ASCII characters, beginning with <ESC>.
WIN_IN_TRANSIT_EVENTS	Enables immediate LOC_MOVE, LOC_WINENTER, and LOC_WINEXIT events. Pick mask only. Off by default.
WIN_LEFT_KEYS	The left function keys, KEY_LEFT(1) — KEY_LEFT(15).
WIN_MOUSE_BUTTONS	Shorthand for MS_RIGHT, MS_MIDDLE and MS_LEFT. Also sets or resets WIN_UP_EVENTS.
WIN_RIGHT_KEYS	The right function keys, KEY_RIGHT(1) — KEY_RIGHT(15).
WIN_TOP_KEYS	The top function keys, KEY_TOP(1) — KEY_TOP(15).
WIN_UP_ASCII_EVENTS	Causes the matching up transitions to normal ASCII events to be reported — if you see an 'a' go down, you'll eventually see the matching 'a' up.
WIN_UP_EVENTS	Causes up transitions to be reported for button and function key events being consumed.

Table 19-12 *Input-Related Window Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
WIN_INPUT_DESIGNEE	int	Window which gets events this window doesn't consume. (Note that the value must be the designee's WIN_DEVICE_NUMBER).
WIN_GRAB_ALL_INPUT	boolean	Window will get all events regardless of location.
WIN_KBD_FOCUS	boolean	Whether or not the window has the keyboard focus.
WIN_KBD_INPUT_MASK	Inputmask *	Window's keyboard inputmask.
WIN_PICK_INPUT_MASK	Inputmask *	Window's pick inputmask.
WIN_CONSUME_KBD_EVENT	short	Window will receive this event.
WIN_IGNORE_KBD_EVENT	short	Window will not receive this event.
WIN_CONSUME_KBD_EVENTS	short list	Null terminated list of events window will receive.
WIN_IGNORE_KBD_EVENTS	short list	Null terminated list of events window will not receive.
WIN_CONSUME_PICK_EVENT	short	Window will receive this pick event.
WIN_IGNORE_PICK_EVENT	short	Window will not receive this pick event.
WIN_CONSUME_PICK_EVENTS	short list	Null terminated list of pick events window will receive.
WIN_IGNORE_PICK_EVENTS	short list	Null terminated list of pick events window will not receive.

Table 19-13 *Menu Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
MENU_ACTION_IMAGE	Pixrect *, action proc	Create image menu item with action proc. Set only.
MENU_ACTION_ITEM	char *, action proc	Create string menu item with action proc. Set only.
MENU_APPEND_ITEM	Menu_item	Append item to end of menu. Set only.
MENU_BOXED	boolean	If TRUE, a single-pixel box will be drawn around every menu item.
MENU_CENTER	boolean	If TRUE, all string items in the menu will be centered. Default: FALSE
MENU_CLIENT_DATA	caddr_t	For client's use.
MENU_COLUMN_MAJOR	boolean	If TRUE, string items in the menu will be sorted in column-major order (like ls(1)) instead of row-major order. Default: FALSE
MENU_CLIENT_DATA	caddr_t	For client's use.
MENU_DESCEND_FIRST	(no value)	For menu_find(). If given, search will be depth first, else search will be "deferred".
MENU_DEFAULT	int	Default menu item as a position.
MENU_DEFAULT_ITEM	Menu_item	Default menu item as opaque handle.
MENU_DEFAULT_SELECTION	enum	Either MENU_SELECTED or MENU_DEFAULT.
MENU_FIRST_EVENT	Event *	The event which was initially passed into menu_show(). Get only. (Note that the event's contents can be modified.)
MENU_FONT	Pixfont *	Menu's font.
MENU_GEN_PROC	(procedure)	Client's function called to generate the menu. Menu_gen_proc(m, op) Menu m; Menu_generate op;
MENU_GEN_PULLRIGHT_IMAGE	Pixrect *, gen proc	Create image menu item with generate proc for pullright. Set only.
MENU_GEN_PULLRIGHT_ITEM	char *, gen proc	Create string menu item with generate proc for pullright. Set only.
MENU_IMAGE_ITEM	Pixrect *, value	Create image menu item with value. Set only.

Table 19-13 *Menu Attributes—Continued*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
MENU_IMAGES	list of Pixrect *	Create multiple image menu items. Set only.
MENU_INITIAL_SELECTION	enum	Either MENU_SELECTED or MENU_DEFAULT.
MENU_INITIAL_SELECTION_EXPANDED	boolean	If TRUE, when the menu pops up, it automatically expands to select the initial selection.
MENU_INITIAL_SELECTION_SELECTED	boolean	If TRUE, menu comes up with its initial selection highlighted. If FALSE, menu comes up with the cursor "standing off" to the left.
MENU_INSERT	int, Menu_item	Insert new item after nth item. Set only.
MENU_INSERT_ITEM	Menu_item, Menu_item	The item given as the second value is inserted after the one given as the first value. Set only.
MENU_ITEM	avlist	Create a menu item inline — avlist same as for menu_create_item(). Set only.
MENU_JUMP_AFTER_NO_SELECTION	boolean	If TRUE, cursor jumps back to its original position after no selection made.
MENU_JUMP_AFTER_SELECTION	boolean	If TRUE, cursor jumps back to its original position after selection made.
MENU_LAST_EVENT	Event *	The last event read by the menu. Get only. Note that the event's contents can be modified.
MENU_LEFT_MARGIN	int	For each string item, margin in addition to MENU_MARGIN on left between menu's border and text. Default: 16.
MENU_MARGIN	int	Margin in pixels around menu items. Default: 1.
MENU_NCOLS	int	Number of columns in menu.
MENU_NITEMS	int	Get only; returns the # of items in the menu.
MENU_NROWS	int	Number of rows in menu.
MENU_NOTIFY_PROC	(procedure)	Client's function called when the user selects a menu item. <pre> caddr_t notify_proc(m, mi) Menu m; Menu_item mi; </pre>

Table 19-13 *Menu Attributes—Continued*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
MENU_NTH_ITEM	int starting from 1.	Get only; returns nth menu item. n is counted starting from 1.
MENU_PARENT	Menu_item	The menu item for which the menu is a pullright. Get only.
MENU_PULLRIGHT_DELTA	int	Number of pixels the user must move the cursor to the right to cause a pullright menu to pop up. Default: 9999.
MENU_PULLRIGHT_IMAGE	Pixrect *, Menu	Create image menu item with pullright. Set only.
MENU_PULLRIGHT_ITEM	char *, Menu	Create string menu item with pullright. Set only.
MENU_REMOVE	int	Remove the nth item. Set only.
MENU_REMOVE_ITEM	Menu_item	Remove the specified item. Set only.
MENU_REPLACE	int, Menu_item	Replace nth item with specified item. Set only.
MENU_REPLACE_ITEM	Menu_item, Menu_item	The item given as first value is replaced with the one given as the second value in the menu (the old item is not replaced in any other menus it may appear in). Set only.
MENU_RIGHT_MARGIN	int	For each string item, margin in addition to MENU_MARGIN on right between menu's border and text.
MENU_SELECTED	int	Last selected item, as a position in menu.
MENU_SELECTED_ITEM	Menu_item	Last selected item, as the item's handle.
MENU_SHADOW	Pixrect *	Pattern for the shadow to be painted behind the menu. If 0, no shadow is painted. Predefined shadow pixrects you can use: menu_gray25_pr, menu_gray50_pr, and menu_gray75_pr.
MENU_STAY_UP	boolean	If TRUE the first <i>click</i> of the Menu button puts up the menu, the second takes it down; in between, the menu stays up. Default: FALSE
MENU_STRINGS	list of char *	Create multiple string menu items. Set only.
MENU_STRING_ITEM	char *, value	Create string menu item with value. Set only.

Table 19-13 *Menu Attributes—Continued*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
MENU_TITLE_IMAGE	Pixrect *	Create image title item. Set only.
MENU_TITLE_ITEM	char *	Create string title item. Set only.
MENU_TYPE	enum	Get only; returns MENU_MENU.
MENU_VALID_RESULT	boolean	Tells whether a zero return value represents a legitimate value.

Table 19-14 *Menu Item Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
MENU_ACTION_IMAGE†	Pixrect *, action proc	Modifies appropriate fields in item. Set only.
MENU_ACTION_ITEM†	char *, action proc	Modifies appropriate fields in item. Set only.
MENU_ACTION_PROC	(procedure)	Client's function called after item has been selected: caddr_t action_proc(menu, menu_item) Menu menu Menu_item menu_item
MENU_APPEND_ITEM†	Menu_item	Append item to end of menu. Set only.
MENU_BOXED†	boolean	If TRUE, a single-pixel box will be drawn around the item.
MENU_CENTER†	boolean	If TRUE, the menu item will be centered on its row in the menu. Only meaningful for menu strings.
MENU_CLIENT_DATA†	caddr_t	For use by the client.
MENU_FEEDBACK	boolean	If FALSE, item is never inverted and is not selectable.
MENU_FONT†	Pixfont *	Item's font.
MENU_GEN_PROC†	(procedure)	Client's procedure called to generate the item.
MENU_GEN_PROC_IMAGE	Pixrect *, (procedure)	Modifies appropriate fields in item. Set only.
MENU_GEN_PROC_ITEM	char *, (procedure)	Modifies appropriate fields in item. Set only.
MENU_GEN_PULLRIGHT	generate proc	Generate proc for the item's pullright.
MENU_GEN_PULLRIGHT_IMAGE†	Pixrect *, (procedure)	Modifies appropriate fields in item. Set only.
MENU_GEN_PULLRIGHT_ITEM†	char *, gen proc	Modifies appropriate fields in item. Set only.
MENU_IMAGE	Pixrect *	Item's image.
MENU_IMAGE_ITEM†	char *, action proc	Modifies appropriate fields in item. Set only.
MENU_INACTIVE	boolean	If TRUE, item is grayed out and not selectable.

† Many of the attributes in this table appeared in the previous table. Menus and menu *items* have many attributes in common. Attributes marked with "†" are also valid for menus, although the effect of the attribute may differ.

Table 19-14 *Menu Item Attributes—Continued*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
MENU_INVERT	boolean	If TRUE, item's display is inverted.
MENU_LEFT_MARGIN†	int	Margin in addition of MENU_MARGIN on left between menu's border and text.
MENU_MARGIN†	int	Margin in pixels around the item.
MENU_PARENT†	Menu	The menu containing the item.
MENU_PULLRIGHT	Menu	Item's pullright menu.
MENU_PULLRIGHT_IMAGE†	Pixrect *, Menu	Modifies appropriate fields in item. Set only.
MENU_PULLRIGHT_ITEM†	char *, Menu	Modifies appropriate fields in item. Set only.
MENU_RELEASE	(no value)	The item will be automatically destroyed when its parent menu is destroyed (default for items created inline).
MENU_RELEASE_IMAGE	(no value)	The string or pixrect associated with the item will be freed when the item is destroyed.
MENU_RIGHT_MARGIN†	int	Margin in addition of MENU_MARGIN on right between menu's border and text.
MENU_SELECTED†	boolean	If TRUE, the item is currently selected.
MENU_STRING†	char *	Item's string.
MENU_STRING_ITEM†	char *, value	Modifies appropriate fields in item. Set only.
MENU_TYPE†	enum	Get only, returns MENU_ITEM.
MENU_VALUE	caddr_t	Item's value.

Table 19-15 *Menu Functions*

<i>Definition</i>	<i>Description</i>
Menu menu_create(attributes) <attribute-list> attributes;	Creates and returns the opaque handle for a walking menu.
Menu_item menu_create_item(attributes) <attribute-list> attributes;	Creates and returns the opaque handle for a single item within a walking menu.
void menu_destroy(menu_object) <Menu or Menu_item> menu_object;	Destroys a menu or menu item.
void menu_destroy_with_proc(menu_object, destroy_proc) <Menu or Menu_item> menu_object; void (*destroy_proc)();	The function supplied as destroy_proc is called before the menu or menu item is destroyed. Arguments: destroy_proc(menu_object, type) <Menu or Menu_item> menu_object; Menu_attribute type; type is MENU_MENU for menus, MENU_ITEM for items.
Menu_item menu_find(menu, attributes) Menu menu; <attribute-list> attributes;	Returns the first menu item in menu meeting the criteria specified in attributes.
caddr_t menu_get(menu_object, attribute[, optional_arg]) <Menu or Menu_item> menu_object; Menu_attribute attributes; caddr_t optional_arg;	Retrieves the value for an attribute of a menu or menu item.
int menu_set(menu_object, attributes) <Menu or Menu_item> menu_object; <attribute-list> attributes;	Sets the value of one or more attributes for a menu or menu item. attributes is a null-terminated attribute list.
caddr_t menu_show(menu, window, event, 0) Menu menu; Window window; Event *event;	Displays the menu, gets a selection from the user, and, by default, returns the value of the item the user has selected. window is the handle of the window over which the menu is displayed; event is the event which causes the menu to come up. The final argument is currently ignored.

Table 19-15 *Menu Functions—Continued*

<i>Definition</i>	<i>Description</i>
<code>caddr_t menu_show_using_fd(menu, fd, event)</code> Menu menu; int fd; Event *event;	Provided for compatibility with SunWindows 2.0. Allows you to display a menu within a window using the windowfd.
<code>caddr_t</code> <code>menu_return_item(menu, menu_item)</code> Menu menu; Menu_item menu_item;	Predefined notify proc which, if given as the value for MENU_NOTIFY_PROC, causes menu_show() to return the handle of the selected item, rather than its value.
<code>caddr_t</code> <code>menu_return_value(menu, menu_item)</code> Menu menu; Menu_item menu_item;	Default notify proc for menus. Causes menu_show() to return the value of the selected item.

Table 19-16 Notifier Functions

Definition	Description
Notify_value notify_default_wait3 (client, pid, status, rusage) Notify_client client; int pid; union wait *status; struct rusage *rusage;	Predefined function you can register with the Notifier via the notify_set_wait3_func() call. Causes the required housekeeping to be performed on the process identified by pid when it dies. See the wait(2) man page for details of the wait and rusage structures.
Notify_error notify_dispatch()	Provided to allow programs which are not notification-based to run in the SunView environment. Called regularly from within the application's main loop to allow the Notifier to go once around its internal loop and dispatch any pending events.
Notify_error notify_do_dispatch()	Called once, before the application's main loop. Enables "implicit dispatching," in which the Notifier dispatches events from within calls to read(2) or select(2) .
Notify_error notify_interpose_destroy_func (client, destroy_func) Notify_client client; Notify_func destroy_func;	Interposes destroy_func() in front of client's destroy event handler.
Notify_error notify_interpose_event_func (client, event_func, type) Notify_client client; Notify_func event_func; Notify_event_type type;	Interposes event_func() in front of client's event handler.
Notify_error notify_itimer_value (client, which, value) Notify_client client; int which; struct itimerval *value;	Returns the current state of an interval timer for client in the structure pointed to by value . The which parameter is either ITIMER_REAL or ITIMER_VIRTUAL .
Notify_value notify_next_destroy_func (client, status) Notify_client client; Destroy_status status;	Calls the next destroy event handler for client . status returns DESTROY_PROCESS_DEATH , DESTROY_CHECKING , or DESTROY_CLEANUP .

Table 19-16 *Notifier Functions—Continued*

<i>Definition</i>	<i>Description</i>
Notify_value notify_next_event_func(client, event, arg, type) Notify_client client; Event *event; Notify_arg arg; Notify_event_type type;	Calls the next event handler for client.
Notify_error notify_no_dispatch()	Prevents the Notifier from dispatching events from within the call to read(2) or select(2).
void notify_perror(s) char *s;	Analogous to the UNIX perror(3) system call. s is printed to stderr, followed by a terse description of notify_errno().
Notify_func notify_set_destroy_func(client, destroy_func) Notify_client client; Notify_func destroy_func;	Registers destroy_func() with the Notifier. destroy_func() will be called when a destroy event is posted to client or when the process receives a SIGTERM signal.
Notify_func notify_set_exception_func(client, exception_func, fd) Notify_client client; Notify_func exception_func; int fd;	Registers the exception handler exception_func() with the Notifier. The only known devices that generate exceptions at this time are stream-based socket connections when an out-of-band byte is available.
Notify_func notify_set_input_func(client, input_func, fd) Notify_client client; Notify_func input_func; int fd;	Registers input_func() with the Notifier. input_func() will be called whenever there is input pending on fd.
Notify_func notify_set_itimer_func(client, itimer_func, which, value, ovalue) Notify_client client; Notify_func itimer_func; int which; struct itimerval *value, *ovalue;	Registers the timeout event handler itimer_func() with the Notifier. The semantics of which, value and ovalue parallel the arguments to setitimer (see the getitimer manual page). which is either ITIMER_REAL or ITIMER_VIRTUAL.

Table 19-16 Notifier Functions— Continued

<i>Definition</i>	<i>Description</i>
Notify_func notify_set_signal_func(client, signal_func, signal, when) Notify_client client; Notify_func signal_func; int signal; Notify_signal_mode when;	Registers the signal event handler <code>signal_func()</code> with the Notifier. <code>signal_func()</code> will be called whenever signal is caught by the Notifier. <code>when</code> can be either <code>NOTIFY_SYNC</code> or <code>NOTIFY_ASYNC</code> . Calling <code>notify_set_signal_func()</code> with a <code>NULL</code> in the place of the <code>signal_func()</code> turns off checking for that signal for that client.
Notify_error notify_start()	Begins dispatching of events by the Notifier.
Notify_error notify_stop()	Terminates dispatching of events by the Notifier.
Notify_func notify_set_output_func(client, output_func, fd) Notify_client client; Notify_func output_func; int fd;	Registers <code>output_func()</code> with the Notifier. <code>output_func()</code> will be called whenever output has been completed on <code>fd</code> .
Notify_func notify_set_wait3_func(client, wait3_func, pid) Notify_client client; Notify_func wait3_func; int pid;	Registers the function <code>wait3_func()</code> with the Notifier. The registered function will be called after the child process identified by <code>pid</code> dies. To do the minimum processing, register the predefined function <code>notify_default_wait3()</code> .
Notify_error notify_veto_destroy(client) Notify_client client;	Called from within a destroy event handler when status is <code>DESTROY_CHECKING</code> and the application does not want to be destroyed.

Table 19-17 *Panel Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
PANEL_ACCEPT_KEYSTROKE	boolean	If TRUE, keystroke events are passed to the panel's PANEL_BACKGROUND_PROC. Default: FALSE.
PANEL_BACKGROUND_PROC	(procedure)	Event handling procedure called when an event falls on the background of the panel. Form: background_proc(panel, event) Panel panel Event *event
PANEL_BLINK_CARET	boolean	If TRUE, the caret blinks. Default: setting of <i>Blink_caret</i> in the <i>Text</i> category of default_sedit.
PANEL_CARET_ITEM	Panel_item	Text item which currently has the caret. Default: first text item.
PANEL_EVENT_PROC	(procedure)	Event handling procedure for panel items. Sets the default for subsequent items created in panel. Form: event_proc(item, event) Panel_item item Event *event
PANEL_FIRST_ITEM	Panel_item	First item in the panel. Get only.
PANEL_ITEM_X_GAP	int	Number of x-pixels between items. Default: 10.
PANEL_ITEM_Y_GAP	int	Number of y-pixels between items. Default: 5.
PANEL_LABEL_BOLD	boolean	If TRUE, item's label is rendered in bold. Sets the default for subsequent items created in panel. Default: FALSE.
PANEL_LAYOUT	Panel_setting	Layout of item's value relative to the label. PANEL_HORIZONTAL (default) or PANEL_VERTICAL.
PANEL_SHOW_MENU	boolean	If TRUE, the menu for the item is enabled. Sets the default for subsequent items created in panel.

Table 19-18 *Generic Panel Item Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
PANEL_ACCEPT_KEYSTROKE	boolean	If TRUE, keystroke events are passed to the item's EVENT_PROC.
PANEL_CLIENT_DATA	caddr_t	For application's use.
PANEL_EVENT_PROC	(procedure)	Event handling procedure for the item.
PANEL_ITEM_RECT	Rect *	Enclosing rectangle for the item. Get only.
PANEL_ITEM_X	int	Left edge of item rectangle. If unspecified and label or value positions are fixed, then set to min of PANEL_LABEL_X and PANEL_VALUE_X. Default: after lowest, rightmost item
PANEL_ITEM_Y	int	top edge of item rectangle. If unspecified and label or value positions are fixed, then set to min of PANEL_LABEL_Y and PANEL_VALUE_Y. Default: previous item's PANEL_ITEM_Y.
PANEL_LABEL_X	int	Left edge of label. If unspecified and value position is fixed, then set to left of PANEL_VALUE_X for horizontal layout, or at PANEL_VALUE_X for vertical layout. Default: PANEL_ITEM_X.
PANEL_LABEL_Y	int	Top edge of label. If unspecified and value position is fixed, then set to PANEL_VALUE_Y for horizontal layout, or above PANEL_VALUE_Y for vertical layout. Default: PANEL_ITEM_Y.
PANEL_LABEL_BOLD	boolean	If TRUE, item's label is rendered in bold. Default: FALSE.
PANEL_LABEL_FONT	Pixfont *	Font for PANEL_LABEL_STRING. Default: WIN_FONT.
PANEL_LABEL_IMAGE	Pixrect *	Image for item's label.
PANEL_LABEL_STRING	char *	String for item's label.
PANEL_LAYOUT	Panel_setting	Layout of item's value relative to the label. PANEL_HORIZONTAL (default) or PANEL_VERTICAL.
PANEL_MENU_CHOICE_FONTS	list of Pixfont *	Font for each menu choice string. Create, set. Default: WIN_FONT.
PANEL_MENU_CHOICE_IMAGES	list of Pixrect *	Image for each menu choice. Create, set. Default: PANEL_CHOICE_IMAGES for choice items, PANEL_LABEL_IMAGE for button items, NULL for other items.
PANEL_MENU_CHOICE_STRINGS	list of char *	String for each menu choice. Create, set. Default: PANEL_CHOICE_STRINGS for choice items, NULL for other items.
PANEL_MENU_CHOICE_VALUES	list of caddr_t	The values returned from the item's menu. Create, set.

Table 19-18 Generic Panel Item Attributes—Continued

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
PANEL_MENU_TITLE_FONT	Pixfont *	Font for PANEL_MENU_TITLE_STRING.
PANEL_MENU_TITLE_IMAGE	Pixrect *	Image for the menu title.
PANEL_MENU_TITLE_STRING	char *	String for the menu title.
PANEL_NEXT_ITEM	Panel_item	Next item in the panel. Get only.
PANEL_NOTIFY_PROC	(procedure)	<p>Function to call when item is selected. Form for button and text items:</p> <pre> notify_proc(item, event) Panel_item item Event *event </pre> <p>Choice and slider items have an additional parameter for the current value:</p> <pre> notify_proc(item, value, event) Panel_item item int value Event *event </pre> <p>For toggle items, the value parameter is of type unsigned int. The type for a text item notify_proc is Panel_setting.</p>
PANEL_PAINT	Panel_setting	Item's painting behavior for panel_set () calls. One of: PANEL_NONE, PANEL_CLEAR, or PANEL_NO_CLEAR.
PANEL_PARENT_PANEL	Panel	The panel which contains the item.
PANEL_SHOW_ITEM	boolean	Whether or not to show the item. Default: TRUE.
PANEL_SHOW_MENU	boolean	If TRUE, the menu for the item is enabled.
PANEL_VALUE_X	int	Left edge of value. If unspecified and label position is fixed, then set to right of PANEL_LABEL_X for horizontal layout, or at PANEL_LABEL_X for vertical layout. Default: after the label.
PANEL_VALUE_Y	int	Top edge of value. If unspecified and label position is fixed, then set to PANEL_LABEL_Y for horizontal layout, or below PANEL_LABEL_Y for vertical layout. Default: PANEL_LABEL_Y.

Table 19-19 *Choice and Toggle Item Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
PANEL_CHOICE_FONTS	list of Pixfont *	Font to use for each choice string. Create, set.
PANEL_CHOICE_IMAGE	int, pixrect *	Image for choice specified by the first argument.
PANEL_CHOICE_IMAGES	list of Pixrect *	Image for each choice. Create, set.
PANEL_CHOICE_STRING	int, char *	String for choice specified by first argument.
PANEL_CHOICE_STRINGS	list of char *	String for each choice. Note that you must specify at least one choice — the least you can specify is a single null string (PANEL_CHOICE_STRINGS, "", 0). Create, set.
PANEL_CHOICE_X	int, int	Second argument is left edge of choice specified by first argument.
PANEL_CHOICE_XS	list of int	Left edge of each choice. Create, set.
PANEL_CHOICE_Y	int, int	Second argument is top edge of choice specified by first argument.
PANEL_CHOICE_YS	list of int	Top edge of each choice. Create, set.
PANEL_CHOICES_BOLD	boolean	If TRUE, choices strings are in bold. Default: FALSE.
PANEL_DISPLAY_LEVEL	Panel_setting	How many choices to display. One of PANEL_NONE, PANEL_CURRENT, or PANEL_ALL. Default: PANEL_ALL.
PANEL_FEEDBACK	Panel_setting	Feedback to give when a choice is selected. One of PANEL_NONE, PANEL_MARKED, PANEL_INVERTED. If PANEL_DISPLAY_LEVEL is PANEL_CURRENT, default is PANEL_NONE, otherwise PANEL_MARKED.
PANEL_LAYOUT	Panel_setting	Layout of the choices: PANEL_HORIZONTAL (default) or PANEL_VERTICAL.
PANEL_MARK_IMAGE	int, Pixrect *	Image to mark choice specified by the first argument when it is selected. Default is push-button image: <images/panel_choice_on.pr>.

Table 19-19 *Choice and Toggle Item Attributes— Continued*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
PANEL_MARK_IMAGES	list of Pixrect *	Image to mark each choice with when selected. Create, set only. Default is push-button image: <images/panel_choice_on.pr>.
PANEL_MARK_X	int, int	Second argument is left edge of choice mark specified by first argument.
PANEL_MARK_XS	list of int	Left edge of each choice mark. Create, set.
PANEL_MARK_Y	int, int	Second argument is top edge of choice mark specified by first argument.
PANEL_MARK_YS	list of int	Top edge of each choice mark. Create, set.
PANEL_MENU_MARK_IMAGE	Pixrect *	Image to mark each menu choice with when selected.
PANEL_MENU_NOMARK_IMAGE	Pixrect *	Image to mark each menu choice with when not selected.
PANEL_NOMARK_IMAGE	int, Pixrect *	Image to mark choice specified by the first argument when it is not selected. Default is push-button image: <images/panel_choice_off.pr>.
PANEL_NOMARK_IMAGES	list of Pixrect *	Image to mark each choice with when not selected. Create, set. Default is push-button image: <images/panel_choice_off.pr>.
PANEL_SHOW_MENU_MARK	boolean	Show or don't show the menu mark for each selected choice. Default: TRUE.
PANEL_TOGGLE_VALUE	int, int	Value of a particular toggle choice. Second argument is value of choice specified by first argument.
PANEL_VALUE	int or unsigned	If item is a choice, value is ordinal position (from 0) of current choice. If item is a toggle, value is a bitmask indicating currently selected choices (e.g., bit 5 is 1 if 5th choice selected).

Table 19-20 *Slider Item Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
PANEL_MIN_VALUE	int	Minimum value of slider. Default: 0.
PANEL_MAX_VALUE	int	Maximum value of the slider. Default: 100.
PANEL_NOTIFY_LEVEL	Panel_setting	When to call the notify function: PANEL_DONE notifies when the select button is released, PANEL_ALL notifies continuously as the select button is dragged. Default: PANEL_DONE.
PANEL_SHOW_RANGE	boolean	Show or don't show the min and max slider values. Default: TRUE.
PANEL_SHOW_VALUE	boolean	Show or don't show integer value of slider. Default: TRUE.
PANEL_SLIDER_WIDTH	int	Width of the slider bar in pixels. Default: 100.
PANEL_VALUE	int	Initial or new value for the item, in the range PANEL_MIN_VALUE to PANEL_MAX_VALUE. Default: PANEL_MIN_VALUE.
PANEL_VALUE_FONT	Pixfont *	Font to use when displaying the value.

Table 19-21 *Text Item Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
PANEL_MASK_CHAR	char	Character used to mask type-in characters. Use the space character for no character echo (caret does not advance). Use the null character to disable masking.
PANEL_NOTIFY_LEVEL	Panel_setting	When to call the notify function. One of PANEL_NONE, PANEL_NON_PRINTABLE, PANEL_SPECIFIED, or PANEL_ALL. Default: PANEL_SPECIFIED (see <i>Text Notification</i>).
PANEL_NOTIFY_STRING	char *	String of characters which trigger notification when typed. Applies only when PANEL_NOTIFY_LEVEL is PANEL_SPECIFIED. Default: \n\r\t (newline, carriage return and tab).
PANEL_VALUE_STORED_LENGTH	int	Max number of characters to store in the value string. Default: 80.
PANEL_VALUE_DISPLAY_LENGTH	int	Max number of characters to display in the panel. Default: 80.
PANEL_VALUE	char *	Initial or new string value for the item.
PANEL_VALUE_FONT	Pixfont *	Font to use for the value string.

Table 19-22 *Panel Functions and Macros*

<i>Definition</i>	<i>Description</i>
<code>panel_accept_key(object, event)</code> <i><Panel or Panel_item></i> object; Event *event;	Action function which tells a text item to accept a keyboard event. Currently ignored by non-text panel items.
<code>panel_accept_menu(object, event)</code> <i><Panel or Panel_item></i> object; Event *event;	Action function which tells an item to display its menu and process the user's selection.
<code>panel_accept_preview(object, event)</code> <i><Panel or Panel_item></i> object; Event *event;	Action function which tells an item to do what it is supposed to do when it is selected. This may include completing feedback initiated by <code>panel_begin_preview()</code> .
Panel_item <code>panel_advance_caret(panel)</code> Panel panel;	Advance the caret to the next text item. If on the last text item, rotate back to the first. Returns the new caret item, or NULL if there are no text items.
Panel_item <code>panel_backup_caret(panel)</code> Panel panel;	Backup the caret to the previous text item. If on the first text item, rotate back to the first. Returns the new caret item, or NULL if there are no text items.
<code>panel_begin_preview(object, event)</code> <i><Panel or Panel_item></i> object; Event *event;	Action function which tells an item to begin any feedback which indicates tentative selection.
Pixrect * <code>panel_button_image(panel, string, width, font)</code> Panel panel; char *string; int width; Pixfont *font;	Creates a standard, button-like image from a string. The string is rendered in font, centered within a double-pixel border width characters wide. If width is too narrow for the string, the border will be expanded to contain the entire string. If font is 0, panel's font is used.
<code>panel_cancel_preview(object, event)</code> <i><Panel or Panel_item></i> object; Event *event;	Action function which tells an item to cancel the feedback initiated by <code>panel_begin_preview()</code> .
Panel_item <code>panel_create_item(panel, item_type, attributes)</code> Panel panel; <i><item type></i> item_type; <i><attribute-list></i> attributes;	Creates and returns the opaque handle to a panel item. item_type is one of: PANEL_MESSAGE, PANEL_BUTTON, PANEL_CHOICE, PANEL_CYCLE, PANEL_TOGGLE, PANEL_TEXT or PANEL_SLIDER. attributes is a null-terminated attribute list.

Table 19-22 *Panel Functions and Macros—Continued*

<i>Definition</i>	<i>Description</i>
<pre>panel_default_handle_event(object, event) <Panel or Panel_item> object; Event *event;</pre>	<p>The default event proc for panel items (PANEL_EVENT_PROC) and for the panel's background (PANEL_BACKGROUND_PROC). Implements the standard event-to-action mapping for the item types.</p>
<pre>panel_destroy_item(item) Panel_item item;</pre>	<p>Destroys item.</p>
<pre>panel_each_item(panel, item) Panel panel; Panel_item item;</pre>	<p>Macro to iterate over each item in a panel. The corresponding macro panel_end_each closes the loop opened by panel_each_item().</p>
<pre>Event * panel_event(panel, event) Panel panel; Event *event;</pre>	<p>Translates the coordinates of event from the space of the panel subwindow to the space of the logical panel (which may be larger and scrollable).</p>
<pre>caddr_t panel_get(item, attribute[, optional_arg]) Panel_item item; Panel_attribute attribute; Panel_attribute optional_arg;</pre>	<p>Retrieve the value of an attribute for item. optional_arg is used for a few attributes which require additional information, such as PANEL_CHOICE_IMAGE, PANEL_CHOICE_STRING, PANEL_CHOICE_X, PANEL_CHOICE_Y, PANEL_MARK_X, PANEL_MARK_Y, PANEL_TOGGLE_VALUE.</p>
<pre>caddr_t panel_get_value(item) Panel_item item;</pre>	<p>A macro, defined as: panel_get(item, PANEL_VALUE)</p>
<pre>panel_paint(panel_object, paint_behavior) <Panel_item or Panel> panel_object; Panel_setting paint_behavior;</pre>	<p>Paints an item or an entire panel. paint_behavior can be either PANEL_CLEAR or PANEL_NO_CLEAR. PANEL_CLEAR causes the area occupied by the panel or item to be cleared prior to painting.</p>
<pre>panel_set(item, attributes) Panel_item item; <attribute-list> attributes;</pre>	<p>Sets the value of one or more panel attributes. attributes is a null-terminated attribute list.</p>
<pre>panel_set_value(item, value) Panel_item item; caddr_t value;</pre>	<p>A macro, defined as: panel_set(item, PANEL_VALUE, value, 0)</p>
<pre>Panel_setting panel_text_notify(item, event) Panel_item item Event *event</pre>	<p>Default notify procedure for panel text items. Causes caret to advance on CR or tab, caret to backup on shift-CR or shift-tab, printable characters to be inserted into item's value, and all other characters to be discarded.</p>

Table 19-22 *Panel Functions and Macros—Continued*

<i>Definition</i>	<i>Description</i>
<code>panel_update_preview(object, event)</code> <Panel or Panel_item> object; Event *event;	Action function which tells the item to update its previewing feedback (e.g. redraw the slider bar for a slider item).
<code>panel_update_scrolling_size(panel)</code> Panel panel;	Updates the scrollbar's notion of the panel's size, so the scrollbar's bubble will be the correct size.
Event * <code>panel_window_event(panel, event)</code> Panel panel; Event *event;	Translates the coordinates of event to the space of the panel subwindow from the space of the logical panel (which may be larger and scrollable).

Table 19-23 *Pixwin Drawing Functions and Macros*

Definition	Description
<pre>pw_batch(pw, n) Pixwin *pw; Pw_batch_type n;</pre>	Tells the batching mechanism to refresh the screen every n display operations.
<pre>pw_batch_off(pw) Pixwin *pw;</pre>	A macro to turn batching off in pw.
<pre>pw_batch_on(pw) Pixwin *pw;</pre>	A macro to turn batching on in pw.
<pre>pw_batchrop(pw, dx, dy, op, items, n) Pixwin *pw; int dx, dy, op, n; struct pr_prpos items[];</pre>	See the <i>Pixrect Reference Manual</i> for a full explanation of this function.
<pre>pw_char(pw, x, y, op, font, c) Pixwin *pw; int x, y, op; Pixfont *font; char c;</pre>	Writes character c into pw using the rasterop op. The left edge and baseline of c will be written at location (x, y).
<pre>pw_close(pw) Pixwin *pw;</pre>	Frees any dynamic storage associated with pw, including its retained memory pixrect, if any.
<pre>pw_copy(dpw, dx, dy, dw, dh, op, spw, sx, sy) Pixwin *dpw, *spw; int op, dx, dy, dw, dh, sx, sy;</pre>	Copies pixels from spw to dpw. Currently spw and dpw must be the same. This routine will cause problems if spw is obscured.
<pre>int pw_get(pw, x, y) Pixwin *pw; int x, y;</pre>	Returns the value of the pixel at (x, y) in pw.
<pre>int pw_get_region_rect(pw, r) Pixwin *pw; Rect *r;</pre>	Retrieves the rectangle occupied by the region pw into the rect pointed to by r.

Table 19-23 *Pixwin Drawing Functions and Macros—Continued*

<i>Definition</i>	<i>Description</i>
<pre> pw_line(pw, x0, y0, x1, y1, brush, tex, op) Pixwin *pw; int x0, y0, x1, y1, op; struct pr_brush *brush; struct pr_texture *tex; </pre>	Draws a solid or textured line between two points with a “brush” of a specified width.
<pre> pw_lock(pw, r) Pixwin *pw; Rect *r; </pre>	Acquires a lock for the user process making the call. <i>r</i> is the rectangle in <i>pw</i> ’s coordinate system that bounds the area to be affected.
<pre> pw_pfsysclose() </pre>	Closes the system font opened with <i>pw_pfsysopen()</i> .
<pre> Pixfont * pw_pfsysopen() </pre>	Opens the system font.
<pre> pw_polygon_2(pw, dx, dy, nbds, npts, vlist, op, spr, sx, sy) Pixwin *pw; int dx, dy, nbds, op, sx, sy; int npts[]; struct pr_pos *vlist; Pixrect *spr; </pre>	Draws a polygon in <i>pw</i> .
<pre> pw_polyline(pw, dx, dy, npts, ptlist, mvlist, brush, tex, op) Pixwin *pw; int dx, dy, npts, op; struct pr_pos *ptlist; u_char *mvlist; struct pr_brush *brush; struct pr_texture *tex; </pre>	Draws multiple lines of a specified width and texture in <i>pw</i> .
<pre> pw_polypoint(pw, dx, dy, npts, ptlist, op) Pixwin *pw; int dx, dy, npts, op; struct pr_pos *ptlist; </pre>	Draws an array of <i>npts</i> points in the pixwin <i>pw</i>
<pre> pw_put(pw, x, y, value) Pixwin *pw; int x, y, value; </pre>	Draws a pixel of value at (<i>x</i> , <i>y</i>) in <i>pw</i> .

Table 19-23 *Pixwin Drawing Functions and Macros—Continued*

<i>Definition</i>	<i>Description</i>
<pre> pw_read(pr, dx, dy, dw, dh, op, pw, sx, sy) Pixwin *pw; int op, dx, dy, dw, dh, sx, sy; Pixrect *pr; </pre>	Reads pixels from the pixwin <code>pw</code> starting at offset (<code>sx</code> , <code>sy</code>), using rasterop <code>op</code> . The pixels are stored in the rectangle (<code>dx</code> , <code>dy</code> , <code>dw</code> , <code>dh</code>) in the pixrect pointed to by <code>pr</code> .
<pre> Pixwin * pw_region(pw, x, y, width, height) Pixwin *pw; int x, y, w, h; </pre>	Creates a new pixwin referring to an area within the existing pixwin <code>pw</code> . The origin of the new region is given by (<code>x</code> , <code>y</code>), the dimensions by <code>width</code> and <code>height</code> .
<pre> pw_replrop(pw, dx, dy, dw, dh, op, pr, sx, sy) Pixwin *pw; int dx, dy, dw, dh, op, sx, sy; Pixrect *pr; </pre>	Replicates a pattern from a pixrect into a pixwin.
<pre> pw_reset(pw) Pixwin *pw; </pre>	Macro which sets <code>pw</code> 's lock count to 0 and releases its lock.
<pre> pw_rop(pw, dx, dy, dw, dh, op, sp, sx, sy) Pixwin *pw; Pixrect *sp; int dx, dy, dw, dh, op, sx, sy; </pre>	Performs the rasterop <code>op</code> from the source pixrect <code>sp</code> to the destination pixwin <code>pw</code> .
<pre> int pw_set_region_rect(pw, r, use_same_pr) Pixwin *pw; Rect *r; unsigned int use_same_pr; </pre>	<p>The position and size of the region <code>pw</code> are set to the rect <code>*r</code>.</p> <p>If <code>use_same_pr</code> is 0 a new retained pixrect is allocated for the region.</p>
<pre> pw_show(pw) Pixwin *pw; </pre>	Macro to refresh the screen while batching, without affecting the batching mode.
<pre> pw_stencil(dpw, dx, dy, dw, dh, op, stpr, stx, sty, spr, sx, sy) Pixwin *dpw; int dx, dy, dw, dh, op, stx, sty, sx, sy; Pixrect *stpr, *spr; </pre>	Like <code>pw_write()</code> , except that the source pixrect <code>spr</code> is written through the stencil pixrect <code>stpr</code> , which functions as a spatial write enable mask. The raster operation <code>op</code> is only applied to destination pixels where the <code>stpr</code> is non-zero; other destination pixels remain unchanged.
<pre> pw_text(pw, x, y, op, font, s) Pixwin *pw; int x, y, op; Pixfont *font; char *s; </pre>	Writes the string <code>s</code> into <code>pw</code> using the rasterop <code>op</code> . The left edge and baseline of the first character in <code>s</code> will appear at coordinates (<code>x</code> , <code>y</code>).

Table 19-23 *Pixwin Drawing Functions and Macros—Continued*

<i>Definition</i>	<i>Description</i>
<pre> pw_traprop(pw, dx, dy, t, op, pr, sx, sy) Pixwin *pw; struct pr_trap t; Pixrect *pr; int dx, dy, op, sx, sy; </pre>	Like <code>pw_rop()</code> , but operating on a trapezon rather than a rectangle.
<pre> pw_ttext(pw, x, y, op, font, s) Pixwin *pw; int x, y, op; Pixfont *font; char *s; </pre>	Like <code>pw_text()</code> except that it writes “transparent” text, i.e. it writes the shape of the letters without disturbing the background behind the letters.
<pre> pw_unlock(pw) Pixwin *pw; </pre>	Decrements the lock count for <code>pw</code> . If the lock count goes to 0, the lock is released.
<pre> pw_vector(pw, x0, y0, x1, y1, op, value) Pixwin *pw; int op, x0, y0, x1, y1, value; </pre>	Draws a vector of pixel <code>value</code> from <code>(x0, y0)</code> to <code>(x1, y1)</code> in <code>pw</code> using rasterop <code>op</code> .
<pre> pw_write(pw, dx, dy, dw, dh, op, pr, sx, sy) Pixwin *pw; int dx, dy, dw, dh, op, sx, sy; Pixrect *pr; </pre>	Writes pixels to <code>pw</code> in the rectangle defined by <code>dx, dy, dw, dh</code> , using rasterop <code>op</code> . Pixels to write are taken from the rectangle with its origin at <code>sx, sy</code> in the source pixmap pointed to by <code>pr</code> . Note: this is an alternative form of <code>pw_rop</code> .
<pre> pw_writebackground(pw, dx, dy, dw, dh, op) Pixwin *pw; int dx, dy, dw, dh, op; </pre>	Writes pixels with value zero into <code>pw</code> using the rasterop <code>op</code> . <code>xd, yd, width</code> and <code>height</code> specify the rectangle in <code>pw</code> which is affected.

Table 19-24 *Pixwin Color Manipulation Functions*

<i>Definition</i>	<i>Description</i>
<pre>pw_blackonwhite(pw, min, max) Pixwin *pw; int min, max;</pre>	Sets the foreground to black, the background to white, for pixwin pw. min and max should be the first and last entries, respectively, in pw's colormap segment.
<pre>pw_cyclecolormap(pw, cycles, index, count) Pixwin *pw; int cycles, index, count;</pre>	Rotates the portion of pw's colormap segment starting at index for count entries, rotating those entries among themselves cycles times.
<pre>pw_dbl_access(pw) Pixwin *pw;</pre>	Resets the window's data structure so that the first frame will be rendered to the background.
<pre>pw_dbl_flip(pw) Pixwin *pw;</pre>	Allows you to flip the display.
<pre>pw_dbl_get(pw, attribute) Pixwin *pw; Pw_dbl_attribute attribute;</pre>	Retrieves the value of the specified attribute.
<pre>pw_dbl_release() Pixwin *pw;</pre>	Signifies the end of double-buffering by the window associated with the pixwin.
<pre>pw_dbl_set(pw, attributes) Pixwin *pw; <attribute-list> attributes;</pre>	Sets the pixwin hardware double-buffering attributes in attributes.
<pre>pw_getattributes(pw, planes) Pixwin *pw; int *planes;</pre>	Retrieves the value of pw's access enable mask into the integer addressed by planes.
<pre>pw_getcmsname(pw, cmsname) Pixwin *pw; char cmsname[CMS_NAMESIZE];</pre>	Copies the colormap segment name of pw into cmsname.
<pre>pw_getcolormap(pw, index, count, red, green, blue) Pixwin *pw; int index, count; unsigned char red[], green[], blue[];</pre>	Retrieves the state of pw's colormap. The count elements of the pixwin's colormap segment starting at index (0 origin) are loaded into the first count values in the three arrays.

Table 19-24 *Pixwin Color Manipulation Functions— Continued*

<i>Definition</i>	<i>Description</i>
<pre>pw_getdefaultcms(cms, map) struct colormapseg *cms; struct cms_map *map;</pre>	Copies the data in the default colormap segment into the data pointed to by <code>cms</code> and <code>map</code> . Before the call, the byte pointers in <code>map</code> should be initialized to arrays of size 256.
<pre>pw_putattributes(pw, planes) Pixwin *pw; int *planes;</pre>	Sets the access enable mask of <code>pw</code> . Only those bits of the pixel corresponding to a 1 in the same bit position of <code>*planes</code> will be affected by <code>pixwin</code> operations.
<pre>pw_putcolormap(pw, index, count, red, green, blue) Pixwin *pw; int index, count; unsigned char red[], green[], blue[];</pre>	Sets the state of <code>pw</code> 's colormap. The <code>count</code> elements of the <code>pixwin</code> 's colormap segment starting at <code>index</code> (0 origin) are loaded from the first <code>count</code> values in the three arrays.
<pre>pw_reversevideo(pw, min, max) Pixwin *pw; int min, max;</pre>	Reverses the foreground and background colors of <code>pw</code> . <code>min</code> and <code>max</code> should be the first and last entries, respectively, in the colormap segment.
<pre>pw_setcmsname(pw, cmsname) Pixwin *pw; char cmsname[CMS_NAMESIZE];</pre>	<code>cmsname</code> is the name that <code>pw</code> will call its window's colormap segment. This call resets the colormap segment to NULL.
<pre>pw_whiteonblack(pw, min, max) Pixwin *pw; int min, max;</pre>	Sets the foreground to white, the background to black, for <code>pw</code> . <code>min</code> and <code>max</code> should be the first and last entries, respectively, in the colormap segment.

Table 19-25 Scrollbar Attributes

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
SCROLL_ABSOLUTE_CURSOR	Cursor	Cursor to display on middle button down. Default: Right triangle if vert., down triangle if horiz.
SCROLL_ACTIVE_CURSOR	Cursor	Cursor to display when cursor is in bar rect. Default: Right arrow if vertical, down arrow if horiz.
SCROLL_ADVANCED_MODE	boolean	Whether notify proc reports all nine motions. Default: FALSE.
SCROLL_BACKWARD_CURSOR	Cursor	Cursor to display on right button down. Default: up arrow if vertical, left arrow if horiz.
SCROLL_BAR_COLOR	Scrollbar_setting	Color of bar, SCROLL_GREY (default) or SCROLL_WHITE.
SCROLL_BAR_DISPLAY_LEVEL	Scrollbar_setting	When bar is displayed. SCROLL_ALWAYS: always displayed SCROLL_ACTIVE: only displayed when cursor is in bar rect SCROLL_NEVER: never displayed Default: SCROLL_ALWAYS.
SCROLL_BORDER	boolean	Whether the scrollbar has a border.
SCROLL_BUBBLE_COLOR	Scrollbar_setting	Color of bubble, SCROLL_GREY (default) or SCROLL_BLACK.
SCROLL_BUBBLE_DISPLAY_LEVEL	Scrollbar_setting	When bubble is displayed. SCROLL_ALWAYS: always displayed SCROLL_ACTIVE: only displayed when cursor is in bar rect SCROLL_NEVER: never displayed Default: SCROLL_ALWAYS.
SCROLL_BUBBLE_MARGIN	int	Margin on each side of bubble in bar. Default: 0.
SCROLL_DIRECTION	Scrollbar_setting	Orientation of bar, SCROLL_VERTICAL (default) or SCROLL_HORIZONTAL.
SCROLL_END_POINT_AREA	int	The distance, in pixels, from the end of the scrollbar that forces a scroll to the beginning (or end) of the file. Default: 6.
SCROLL_FORWARD_CURSOR	Cursor	Cursor to display on left button down. Default: down arrow if vertical, right arrow if horiz.
SCROLL_GAP	int	Gap between lines. Default: current value of SCROLL_MARGIN.

Table 19-25 *Scrollbar Attributes—Continued*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
SCROLL_HEIGHT	int	r_height for scrollbar's rect.
SCROLL_LAST_VIEW_START	int	Offset of view into object prior to scroll. Get only.
SCROLL_LEFT	int	r_left for scrollbar's rect.
SCROLL_LINE_HEIGHT	int	Number of pixels from one line to the next. Default: 0.
SCROLL_MARGIN	int	Top margin after scroll, if SCROLL_NORMALIZE TRUE. Default: 4.
SCROLL_MARK	int	Position (in client units) undo will go to. Initial value: 0.
SCROLL_NOTIFY_CLIENT	caddr_t	Used by Notifier.
SCROLL_NORMALIZE	boolean	Whether the client wants normalized scrolling. Default: TRUE.
SCROLL_OBJECT	caddr_t	Pointer to the scrollable object.
SCROLL_OBJECT_LENGTH	int	Length of scrollable object, in client units. Default: 0. (Value must be > 0).
SCROLL_PAGE_BUTTONS	boolean	Whether the scrollbar has page buttons. Default: TRUE.
SCROLL_PAGE_BUTTON_LENGTH	int	Length in pixels of page buttons. Default: 15.
SCROLL_PAINT_BUTTONS_PROC	(procedure)	Procedure which paints page buttons: paint_buttons_proc(scrollbar) Scrollbar scrollbar; Setting the value to NULL resets it to the default button painting procedure.
SCROLL_PIXWIN	Pixwin *	Pixwin for scrollbar to write to.
SCROLL_PLACEMENT	Scrollbar_setting	Placement of the bar. SCROLL_WEST: vertical bar on left edge SCROLL_EAST: vertical bar on right edge SCROLL_NORTH: horizontal bar on top edge SCROLL_SOUTH: horizontal bar on bottom edge Default: SCROLL_WEST or SCROLL_NORTH.
SCROLL_RECT	Rect *	Rect for scrollbar, including buttons.

Table 19-25 Scrollbar Attributes—Continued

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
SCROLL_REPEAT_TIME	int	The interval, in tenths of a second, that scrolling repeats in. This attribute is used only for the initial pressing down of the mouse. A value of 0 disables repeat scrolling. Default: 10.
SCROLL_REQUEST_MOTION	Scroll_motion	Scrolling motion requested by user.
SCROLL_REQUEST_OFFSET	int	Pixel offset of scrolling request into scrollbar. Default: 0.
SCROLL_THICKNESS	int	Thickness of bar. Default: 14.
SCROLL_TO_GRID	boolean	Whether the client wants scrolling aligned to multiples of SCROLL_LINE_HEIGHT. Default: FALSE.
SCROLL_TOP	int	r_top for scrollbar's rect.
SCROLL_VIEW_LENGTH	int	Length of viewing window, in client units. Default: 0.
SCROLL_VIEW_START	int	Current offset into scrollable object (client units). (Value must be > 0). Default: 0.
SCROLL_WIDTH	int	r_width for scrollbar's rect.

Table 19-26 *Scrollbar Functions*

<i>Definition</i>	<i>Description</i>
<pre>Scrollbar scrollbar_create(attributes) <attribute-list> attributes;</pre>	Creates and returns the opaque handle to a scrollbar.
<pre>int scrollbar_destroy(scrollbar) Scrollbar scrollbar;</pre>	Destroys scrollbar.
<pre>caddr_t scrollbar_get(scrollbar, attribute) Scrollbar scrollbar; Scrollbar_attribute attribute;</pre>	Retrieves the value for an attribute of scrollbar.
<pre>int scrollbar_set(scrollbar, attributes) Scrollbar scrollbar; <attribute-list> attributes;</pre>	Sets the value for one or more attributes of scrollbar. attributes is a null-terminated attribute list.
<pre>void scrollbar_scroll_to(scrollbar, new_view_start) Scrollbar scrollbar; long new_view_start;</pre>	For programmatic scrolling. Effect is as if the user had requested a scroll to new_view_start in the subwindow to which scrollbar is attached.
<pre>int scrollbar_paint(scrollbar) Scrollbar scrollbar;</pre>	Paints those portions of scrollbar (page buttons, bar proper, and bubble) which have been modified since they were last painted.
<pre>int scrollbar_paint_clear(scrollbar) Scrollbar scrollbar;</pre>	Clears and repaints all portions of scrollbar.
<pre>int scrollbar_clear_bubble(scrollbar) Scrollbar scrollbar;</pre>	Clears the bubble in scrollbar.
<pre>int scrollbar_paint_bubble(scrollbar) Scrollbar scrollbar;</pre>	Paints the bubble in scrollbar.

Table 19-27 *Text Subwindow Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
TEXTSW_ADJUST_IS_PENDING_DELETE	boolean	When TRUE, adjusting a selection causes the selection to be pending-delete. Default: FALSE.
TEXTSW_AGAIN_RECORDING	boolean	When FALSE, changes made to the textsw are not repeated when user invokes AGAIN. By disabling when not needed (e.g. for program-driven error logs) you can reduce memory overhead. Default: TRUE.
TEXTSW_AUTO_INDENT	boolean	When TRUE, a new line is automatically indented to match the previous line. Default: FALSE.
TEXTSW_AUTO_SCROLL_BY	int	Number of lines to scroll when type-in moves insert point below the view. Default: 1. Create, get.
TEXTSW_BLINK_CARET	boolean	Determines whether the caret blinks. Default: TRUE.
TEXTSW_BROWSING	boolean	When TRUE, prevents editing of the displayed text. If another file is loaded in, browsing stays on. Default: FALSE.
TEXTSW_CHECKPOINT_FREQUENCY	int	Number of edits between checkpoints. Set to 0 to disable checkpointing. Default: 0.
TEXTSW_CLIENT_DATA	char *	Pointer to arbitrary client data. Default: NULL.
TEXTSW_CONFIRM_OVERWRITE	boolean	A request to write to an existing file will require user confirmation. Default: TRUE.
TEXTSW_CONTENTS	char *	<p>Contents of text subwindow. Default: NULL.</p> <p>For create and set, specifies the initial contents for non-file textsw. Get needs additional parameters:</p> <p style="padding-left: 20px;">window_get(textsw, TEXTSW_CONTENTS, pos, buf, buf_len)</p> <p>Return value is next position to read at.</p> <p>buf[0...buf_len-1] is filled with the characters from textsw beginning at index pos, and is null-terminated only if there were too few characters to fill the buffer.</p>
TEXTSW_CONTROL_CHARS_USE_FONT	boolean	If FALSE, control characters always display as an up arrow followed by a character, instead of whatever glyph is in the current font. Default: FALSE.
TEXTSW_DISABLE_CD	boolean	Stops textsw from changing current working directory (and grays out the associated items in the menu). Default: FALSE.

Table 19-27 *Text Subwindow Attributes— Continued*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
TEXTSW_DISABLE_LOAD	boolean	Prevents files being loaded into the textsw (and grays out the associated items in the menu). Default: FALSE.
TEXTSW_EDIT_COUNT	int	Monotonically incrementing count of the number of edits made to the textsw. Get.
TEXTSW_FILE	char *	File to load. Default: NULL. Create, set.
TEXTSW_FILE_CONTENTS	char *	initializes the text subwindow contents from a file yet still edits the contents in memory.
TEXTSW_FIRST	int	Zero-based index of first displayed character.
TEXTSW_FIRST_LINE	int	Zero-based index of first displayed line.
TEXTSW_HISTORY_LIMIT	int	Number of user action sequences that can be undone. Default: 50. Create, get.
TEXTSW_IGNORE_LIMIT	int	Number of edits textsw allows before vetoing destroy. Valid values are 0, meaning destroy will be vetoed if any edits have been done, and TEXTSW_INFINITY, meaning destroy will never be vetoed. Default: 0.
TEXTSW_INSERT_FROM_FILE	string	inserts the contents of a file into a text subwindow at the current insertion point.
TEXTSW_INSERT_MAKES_VISIBLE	Textsw_enum	Controls whether insertion causes repositioning to make inserted text visible. Possible values are TEXTSW_ALWAYS, TEXTSW_NEVER and TEXTSW_IF_AUTO_SCROLL. Default: TEXTSW_IF_AUTO_SCROLL.
TEXTSW_INSERTION_POINT	Textsw_index	Index of the current insertion point. Get, set.
TEXTSW_LEFT_MARGIN	int	Number of pixels in the margin on left. Default: 4. Create, get.
TEXTSW_LENGTH	int	Length of the textsw's contents. Get only.
TEXTSW_LINE_BREAK_ACTION	Textsw_enum	Determines how the textsw treats file lines too big to fit on one display line. Possible values are either TEXTSW_CLIP or TEXTSW_WRAP_AT_CHAR. Default: TEXTSW_WRAP_AT_CHAR. Create, set.
TEXTSW_LOWER_CONTEXT	int	Minimum # of lines to maintain between insertion point and the bottom of view. Used by auto scrolling when type-in would disappear off bottom of view. -1 means defeat auto scrolling. Default: 2.

Table 19-27 Text Subwindow Attributes— Continued

Attribute	Value Type	Description
TEXTSW_MEMORY_MAXIMUM	int	How much memory to use when not editing files. This attribute only takes effect at textsw window creation time or after the window has been reset via textsw_reset(). The lower bound of the attribute is 1000 bytes which is silently enforced. Default: 20,000 bytes. (If a great deal of text will be inserted into the text subwindow, either by the program or the user, you may need to increase this.)
TEXTSW_MENU	Menu	The text subwindow's menu. Get, set.
TEXTSW_MODIFIED	boolean	Whether or not the textsw has been modified. Get only.
TEXTSW_MULTI_CLICK_SPACE	int	Max # of pixels that can be between successive mouse clicks and still have the clicks be considered a multi-click. Default: 3.
TEXTSW_MULTI_CLICK_TIMEOUT	int	Max # of milliseconds that can be between successive mouse clicks and still have the clicks be considered a multi-click. Default: 390.
TEXTSW_NOTIFY_PROC	(procedure)	Notify procedure. Form is: <pre>void notify_proc(textsw, avlist) Textsw textsw Attr_avlist avlist</pre> Default: NULL, meaning standard procedure.
TEXTSW_READ_ONLY	boolean	When TRUE, prevents editing of the displayed text. If another file is loaded in, READ_ONLY is turned off again. Default: FALSE.
TEXTSW_SCROLLBAR	Scrollbar	Scrollbar to use for text subwindow scrolling. NULL means no scrollbar. Default: A scrollbar with default attributes. Note: text subwindow has a scrollbar by default, so you would only use this to get no scrollbar, or to get the scrollbar handle.
TEXTSW_STATUS	Textsw_status *	If set, specifies the address of a variable of type Textsw_status into which a value is written that reflects what happened during the call to window_create(). (For possible values, see the <i>Textsw_status Values</i> table).
TEXTSW_STORE_CHANGES_FILE	boolean	If TRUE, Store changes the file being edited to that named as the target of the Store. If FALSE, Store does not affect which file is being edited. Default: TRUE.
TEXTSW_STORE_SELF_IS_SAVE	boolean	Causes textsw to interpret a Store to the name of the current file as a Save. Default: FALSE. Create, get.

Table 19-27 *Text Subwindow Attributes— Continued*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
TEXTSW_UPDATE_SCROLLBAR	(no value)	Causes text subwindow to update the bubble in the scrollbar. Set only — get returns NULL.
TEXTSW_UPPER_CONTEXT	int	Min # of lines to maintain between the start of the selection and top of view. -1 means to defeat the normal actions. Default: 2.

Table 19-28 Textsw_action Attributes

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
TEXTSW_ACTION_CAPS_LOCK	boolean	The user pressed the CAPS-lock function key to change the setting of the CAPS-lock (it is initially 0, meaning off).
TEXTSW_ACTION_CHANGED_DIRECTORY	char *	The current working directory for the process has been changed to the directory named by the provided string value.
TEXTSW_ACTION_EDITED_FILE	char *	The file named by the provided string value has been edited. Appears once per session of edits (see below).
TEXTSW_ACTION_EDITED_MEMORY	none	monitors whether an empty text subwindow has been edited.
TEXTSW_ACTION_FILE_IS_READONLY	char *	The file named by the provided string value does not have write permission.
TEXTSW_ACTION_LOADED_FILE	char *	The text subwindow is being used to view the file named by the provided string value.
TEXTSW_ACTION_TOOL_CLOSE	(no value)	The frame containing the text subwindow should become iconic.
TEXTSW_ACTION_TOOL_DESTROY	Event *	The tool containing the text subwindow should exit, without checking for a veto from other subwindows. The value is the user action that caused the destroy.
TEXTSW_ACTION_TOOL_QUIT	Event *	The tool containing the text subwindow should exit normally. The value is the user action that caused the exit.
TEXTSW_ACTION_TOOL_MGR	Event *	The tool containing the text subwindow should do the window manager operation associated with the provided event value.
TEXTSW_ACTION_USING_MEMORY	(no value)	The text subwindow is being used to edit a string stored in primary memory, not a file.

Table 19-29 Textsw_status Values

<i>Value</i>	<i>Description</i>
TEXTSW_STATUS_OKAY	The operation encountered no problems.
TEXTSW_STATUS_BAD_ATTR	The attribute list contained an illegal or unrecognized attribute.
TEXTSW_STATUS_BAD_ATTR_VALUE	The attribute list contained an illegal value for an attribute, usually an out of range value for an enumeration.
TEXTSW_STATUS_CANNOT_ALLOCATE	A call to <code>calloc(2)</code> or <code>malloc(2)</code> failed.
TEXTSW_STATUS_CANNOT_OPEN_INPUT	The specified input file does not exist or cannot be accessed.
TEXTSW_STATUS_CANNOT_INSERT_FROM_FILE	The operation encountered a problem when trying to insert from file.
TEXTSW_STATUS_OUT_OF_MEMORY	The operation ran out of memory while editing in memory.
TEXTSW_STATUS_OTHER_ERROR	The operation encountered a problem not covered by any of the other error indications.

Table 19-30 Text Subwindow Functions

Definition	Description
Textsw_mark textsw_add_mark(textsw, position, flags) Textsw textsw; Textsw_index position; unsigned flags;	Adds a new mark at position. flags can be either TEXTSW_MARK_DEFAULTS or TEXTSW_MARK_MOVE_AT_INSERT.
int textsw_append_file_name(textsw, name) Textsw textsw; char *name;	Returns 0 if textsw is editing a file, and if so appends the name of the file at the end of name.
Textsw_index textsw_delete(textsw, first, last_plus_one) Textsw textsw; Textsw_index first, last_plus_one;	Returns 0 if the operation fails. Removes the span of characters beginning with first, and ending one before last_plus_one.
Textsw_index textsw_edit(textsw, unit, count, direction) Textsw textsw; unsigned unit, count, direction;	Returns 0 if the operation fails. Erases a character, word or line, depending on whether unit is SELN_LEVEL_FIRST, SELN_LEVEL_FIRST+1, or SELN_LEVEL_LINE. If direction is 0, characters after the insertion point are affected, otherwise characters before the insertion point are affected. The operation will be done count times.
Textsw_index textsw_erase(textsw, first, last_plus_one) Textsw textsw; Textsw_index first, last_plus_one;	Returns 0 if the operation fails. Equivalent to textsw_delete(), but does not affect the global shelf.
void textsw_file_lines_visible(textsw, top, bottom) Textsw textsw; int *top, *bottom;	Fills in top and bottom with the file line indices of the first and last file lines being displayed in textsw.
int textsw_find_bytes(textsw, first, last_plus_one, buf, buf_len, flags) Textsw textsw; Textsw_index *first, *last_plus_one; char *buf; unsigned buf_len, flags;	Beginning at the position addressed by first, searches for the pattern specified by buf of length buf_len. Searches forwards if flags is 0, else searches backwards. Returns -1 if no match, else matching span placed in indices addressed by first and last_plus_one.

Table 19-30 *Text Subwindow Functions—Continued*

<i>Definition</i>	<i>Description</i>
Textsw_index textsw_find_mark(textsw, mark) Textsw textsw; Textsw_index mark;	Returns the current position of mark. If this operation fails, it will return TEXTSW_INFINITY.
Textsw textsw_first(textsw) Textsw textsw;	Returns the first view into textsw.
Textsw_index textsw_index_for_file_line(textsw, line) Textsw textsw; int line;	Returns the character index for the first character in the line given by line. If this operation fails, it will return TEXTSW_CANNOT_SET.
Textsw_index textsw_insert(textsw, buf, buf_len) Textsw textsw; char *buf; int buf_len;	Inserts characters in buf into textsw at the current insertion point. The number of characters actually inserted is returned — this will equal buf_len unless there was a memory allocation failure. If there was a failure, it will return 0.
textsw_match_bytes(textsw, first, last_plus_one, start_sym, start_sym_len, end_sym, end_sym_len, field_flag) Textsw textsw; Textsw_index *first, *last_plus_one; char *start_sym, *end_sym; int start_sym_len, end_sym_len; unsigned field_flag;	Searches for a block of text in the textsw's contents which starts with characters matching start_sym and ends with characters matching end_sym. This function places the starting index of the matching block in first and its ending index in last.
Textsw textsw_next(textsw) Textsw textsw;	Returns the next view in the set of views into textsw.
void textsw_normalize_view(textsw, position) Textsw textsw; Textsw_index position;	Repositions the text so that the character at position is visible and at the top of the subwindow.
void textsw_possibly_normalize(textsw, position) Textsw textsw; Textsw_index position;	If the character at position is already visible, this function does nothing. If it is not visible, it repositions the text so that it is visible and at the top of the subwindow.

Table 19-30 Text Subwindow Functions—Continued

Definition	Description
<pre>void textsw_remove_mark(textsw, mark) Textsw textsw; Textsw_mark mark;</pre>	Removes an existing mark from textsw.
<pre>Textsw_index textsw_replace_bytes(textsw, first, last_plus_one, buf, buf_len) Textsw textsw; Textsw_index first; char *buf; unsigned buf_len;</pre>	Replaces the character span from first to last_plus_one by the characters in buf. last_plus_one. The return value is the net number of bytes inserted. The number is negative if the original string is longer than the one that replaces it. If this operation fails, it will return a value of 0.
<pre>void textsw_reset(textsw, x, y) Textsw textsw; int x, y;</pre>	Discards edits performed on the contents of textsw. If needed, a message box will be displayed at x, y.
<pre>unsigned textsw_save(textsw, x, y) Textsw textsw; int x, y;</pre>	Saves any edits made to the file currently loaded into textsw. If needed, a message box will be displayed at x, y.
<pre>int textsw_screen_line_count(textsw) Textsw textsw;</pre>	Returns the number of screen lines in textsw.
<pre>void textsw_scroll_lines(textsw, count) Textsw textsw; int count;</pre>	Moves the text up or down by count lines. If count is positive, then the text is scrolled up on the screen, (forward in the file); if negative, the text is scrolled down, (backward in the file).
<pre>void textsw_set_selection(textsw, first, last_plus_one, type) Textsw textsw; Textsw_index first, last_plus_one; unsigned type;</pre>	Sets the selection to begin at first and include all characters up to last_plus_one.

Table 19-30 *Text Subwindow Functions— Continued*

<i>Definition</i>	<i>Description</i>
<code>unsigned textsw_store_file(textsw, filename, x, y) Textsw textsw; char *filename; int x, y;</code>	Stores the contents of <code>textsw</code> to the file named by <code>filename</code> . If needed, a message box will be displayed at <code>x, y</code> .

Table 19-31 *TTY Subwindow Attributes*

<i>Attribute</i>	<i>Type</i>	<i>Description</i>
TTY_ARGV	char **	Argument vector: name of the program running in the tty subwindow, followed by arguments for that program.
TTY_CONSOLE	boolean	If TRUE, tty subwindow is console. Set only. Default: FALSE.
TTY_PAGE_MODE	boolean	If TRUE, output will stop after each page. Default: FALSE.
TTY_QUIT_ON_CHILD_DEATH	boolean	If TRUE, window_done() is called on the subwindow when its child terminates. Set only. Default: FALSE.

Table 19-32 *TTY Subwindow Functions*

<i>Definition</i>	<i>Description</i>
<pre> int ttysw_input(tty, buf, len) Tty tty; char *buf; int len; </pre>	Appends len number of characters from buf onto tty's <i>input</i> queue. It returns the number of characters accepted.
<pre> int ttysw_output(tty, buf, len) Tty tty; char *buf; int len; </pre>	Appends len number of characters from buf onto tty's <i>output</i> queue, i.e. they are sent through the terminal emulator to the TTY. It returns the number of characters accepted.

Table 19-33 TTY Subwindow Special Escape Sequences

<i>Escape Sequence</i> ¹⁰³	<i>Description</i>
<code>\E[1t</code>	open frame.
<code>\E[2t</code>	close frame.
<code>\E[3t</code>	move frame with interactive feedback.
<code>\E[3;TOP;LEFTt</code>	move frame to location specified by (TOP,LEFT).
<code>\E[4t</code>	resize frame with interactive feedback.
<code>\E[4;WIDTH;HEIGHTt</code>	resize frame to WIDTH and HEIGHT.
<code>\E[5t</code>	expose.
<code>\E[6t</code>	hide.
<code>\E[7t</code>	redisplay.
<code>\E[8;ROWS;COLSt</code>	resize frame so its width and height are ROWS and COLS.
<code>\E[11t</code>	report if frame is open or closed by sending <code>\[1t</code> or <code>\[2t</code> , respectively.
<code>\E[13t</code>	report frame's position by sending the <code>\E[3;TOP;LEFTt</code> sequence.
<code>\E[14t</code>	report frame's size in pixels by sending the <code>\E[3;WIDTH;HEIGHTt</code> sequence.
<code>\E[18t</code>	report frame's size in characters by sending the <code>\E[8;ROWS;COLSt</code> sequence.
<code>\E[20t</code>	report the frame icon's label by sending the <code>\E[Llabel\E\</code> sequence.
<code>\E[21t</code>	report frame's label by sending the <code>\E]llabel\E\</code> sequence.
<code>\E]ltext\E\</code>	set frame's label to <i>text</i> .
<code>\E]Ifile\E\</code>	set frame's icon to the icon contained in <i>file</i> .
<code>\E]Llabel\E\</code>	set icon's label to <i>label</i> .
<code>\E[>OPT1; . . .OPTnh</code>	turn requested options on. The only currently defined option is 1, for TTY_PAGE_MODE.
<code>\E[>OPT1; . . .OPTnk</code>	turn requested options off.

Table 19-33 *TTY Subwindow Special Escape Sequences—Continued*

<i>Escape Sequence</i> ¹⁰³	<i>Description</i>
<code>\E[>OPT1; . . .OPTn1</code>	report current option settings by sending <code>\E[>OPTx1</code> or <code>\E>OPTn</code> for each option <i>x</i> .

¹⁰³ In this table “\E” denotes the <ESC> character, as it does in *termcap*.

Table 19-34 *Window Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
WIN_BELOW	Window	Causes the window to be laid out below window given as the value.
WIN_BOTTOM_MARGIN	int	Margin at bottom of window.
WIN_CLIENT_DATA	caddr_t	Client's private data — for your use.
WIN_COLUMNS	int	Window's width (including left and right margins) in columns.
WIN_COLUMN_GAP	int	Gap between columns in the window.
WIN_COLUMN_WIDTH	int	Width of a column in the window.
WIN_CONSUME_KBD_EVENT	short	Window will receive this event.
WIN_CONSUME_KBD_EVENTS	list of short	Null terminated list of events window will receive. Create, set.
WIN_CONSUME_PICK_EVENT	short	Window will receive this pick event.
WIN_CONSUME_PICK_EVENTS	list of short	Null terminated list of pick events window will receive. Create, set.
WIN_CURSOR	Cursor	The window's cursor. Note: the pointer returned by <code>window_get()</code> points to per-process static storage.
WIN_DEVICE_NAME	char *	UNIX device name associated with window, consisting of a string and numeric part, e.g. <code>win10</code> . Get only.
WIN_DEVICE_NUMBER	int	Numeric component of device name. Get only.
WIN_ERROR_MSG	char *	Error message to print before <code>exit(1)</code> . Create only.
WIN_EVENT_PROC	(procedure)	Client's callback procedure which receives input events: Notify_value event_proc(window, event, arg) Window window; Event *event; caddr_t arg;
WIN_EVENT_STATE	short	Gets the state of the specified event code. For buttons and keys, zero means "up," non-zero means "down." Get only.
WIN_FD	int	The UNIX file descriptor for the window. Get only.
WIN_FIT_HEIGHT	int	Causes window to fit its contents in the height dimension, leaving a margin specified by the value given.

Table 19-34 Window Attributes—Continued

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
WIN_FIT_WIDTH	int	Causes window to fit its contents in the width dimension, leaving a margin specified by the value given.
WIN_FONT	Pixfont *	The window's font. Notes for the current release: tty subwindows don't use WIN_FONT. Frames don't use WIN_FONT to render their labels; however, they do use WIN_FONT in calculating WIN_COLUMNS and WIN_ROWS. Setting WIN_FONT does <i>not</i> cause the default system font to be set.
WIN_GRAB_ALL_INPUT	boolean	Window will get all events regardless of location.
WIN_HEIGHT	int	Window's height in pixels. Value of WIN_EXTEND_TO_EDGE causes subwindow to extend to bottom edge of frame. Default: WIN_EXTEND_TO_EDGE.
WIN_HORIZONTAL_SCROLLBAR	Scrollbar	Horizontal scrollbar.
WIN_IGNORE_KBD_EVENT	short	Window will not receive this event.
WIN_IGNORE_KBD_EVENTS	list of short	Null terminated list of events window will not receive. Create, set.
WIN_IGNORE_PICK_EVENT	short	Window will not receive this pick event.
WIN_IGNORE_PICK_EVENTS	list of short	Null terminated list of pick events window will not receive. Create, set.
WIN_INPUT_DESIGNEE	int	Window which gets events this window doesn't consume. (Note that the value must be the WIN_DEVICE_NUMBER of the designee).
WIN_KBD_FOCUS	boolean	Whether or not the window has the keyboard focus.
WIN_KBD_INPUT_MASK	Inputmask *	Window's keyboard inputmask. Note: the pointer returned by window_get () points to per-process static storage.
WIN_LEFT_MARGIN	int	Margin at left of window.
WIN_MENU	Menu	Window's menu. Note: In the current release this doesn't work for panels or tty subwindows.
WIN_MOUSE_XY	int, int	Mouse's position within the window. Set only.
WIN_NAME	char *	Name of window (currently unused by SunView).
WIN_OWNER	Window	Owner of window. Get only.
WIN_PERCENT_HEIGHT	int	Sets a subwindow's height as a percentage of the frame's height.

Table 19-34 *Window Attributes—Continued*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
WIN_PERCENT_WIDTH	int	Sets a subwindow's width as a percentage of the frame's width.
WIN_PICK_INPUT_MASK	Inputmask *	Window's pick inputmask. Note: the pointer returned by <code>window_get ()</code> points to per-process static storage.
WIN_PIXWIN	Pixwin *	The window's pixwin. Get only.
WIN_RECT	Rect *	Rect of the window. For frames, same as <code>FRAME_OPEN_RECT</code> . Note: the pointer returned by <code>window_get ()</code> for this attribute points to per-process static storage.
WIN_RIGHT_MARGIN	int	Margin at right of window.
WIN_RIGHT_OF	Window	Causes the window to be laid out just to the right of the window given as the value.
WIN_ROW_GAP	int	Gap between rows in the window.
WIN_ROW_HEIGHT	int	Height of a row in the window.
WIN_ROWS	int	Window's height (including top and bottom margins) in rows.
WIN_SCREEN_RECT	Rect *	Rect of the screen containing the window. Get only. Note: the pointer returned by <code>window_get ()</code> for this attribute points to per-process static storage.
WIN_SHOW	boolean	Causes the window to be displayed or undisplayed.
WIN_TOP_MARGIN	int	Margin at top of window.
WIN_TYPE	Window_type	Type of window. One of <code>FRAME_TYPE</code> , <code>PANEL_TYPE</code> , <code>CANVAS_TYPE</code> , <code>TEXTSW_TYPE</code> or <code>TTY_TYPE</code> . Get only.
WIN_VERTICAL_SCROLLBAR	Scrollbar	Vertical scrollbar.
WIN_WIDTH	int	Window's width in pixels. Value of <code>WIN_EXTEND_TO_EDGE</code> causes subwindow to extend to right edge of frame. Default: <code>WIN_EXTEND_TO_EDGE</code> .
WIN_X	int	x position of window, relative to owner.
WIN_Y	int	y position of window, relative to owner.

Table 19-35 *Frame Attributes*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
FRAME_ARGS	int, char **	Interpret command line arguments. Strips -W command-line frame arguments out of argv. Create only.
FRAME_ARGC_PTR_ARGV	int *, char **	Interpret command line args. Strips -W command-line frame arguments out of argv, and decrements argc accordingly. Create only.
FRAME_BACKGROUND_COLOR	struct singlecolor *	Background color.
FRAME_CLOSED	boolean	Whether frame is currently closed.
FRAME_CLOSED_RECT	Rect *	Frame's rect when closed.
FRAME_CMDLINE_HELP_PROC	(procedure)	Called when user types the command-line argument -WH. Default: <pre>frame_cmdline_help(program_name) char *program_name;</pre>
FRAME_CURRENT_RECT	Rect *	Returns either FRAME_OPEN_RECT or FRAME_CLOSED_RECT, depending on the value of FRAME_CLOSED. Note: in the current release, there is a bug in the behavior of FRAME_CURRENT_RECT for subframes. It is set relative to the owner frame, but it is retrieved relative to the screen.
FRAME_DEFAULT_DONE_PROC	(procedure)	Default value of FRAME_DONE_PROC. Get only. The default procedure is to set the subframe to WIN_SHOW, FALSE.
FRAME_DONE_PROC	(procedure)	Client's proc called when user chooses 'Done' from subframe's menu: <pre>done_proc(frame) Frame frame;</pre>
FRAME_EMBOLDEN_LABEL	boolean	If TRUE, frame's label is rendered in bold.
FRAME_FOREGROUND_COLOR	struct singlecolor *	Foreground color.
FRAME_ICON	Icon	The frame's icon.
FRAME_INHERIT_COLORS	boolean	If TRUE, colormap of frame is inherited by subwindows.
FRAME_LABEL	char *	The frame's label.
FRAME_NO_CONFIRM	boolean	Set to TRUE before destroying a frame to defeat confirmation. Set only.
FRAME_NTH_SUBFRAME	int	Returns frame's nth (from 0) subframe. Get only.

Table 19-35 *Frame Attributes—Continued*

<i>Attribute</i>	<i>Value Type</i>	<i>Description</i>
FRAME_NTH_SUBWINDOW	int	Returns frame's nth (from 0) subwindow. Get only.
FRAME_NTH_WINDOW	int	Returns frame's nth (from 0) window, regardless of whether the window is a frame or a subwindow. Get only.
FRAME_OPEN_RECT	Rect *	Frame's rect when open.
FRAME_SHOW_LABEL	boolean	Whether the label is shown. Default: TRUE for base frames, FALSE for subframes.
FRAME_SUBWINDOWS_ADJUSTABLE	boolean	User can move subwindow boundaries. Default: TRUE.

Table 19-36 Window Functions and Macros

<i>Definition</i>	<i>Description</i>
<pre>void window_bell(win) Window win;</pre>	<p>Queries the user defaults database to see if the user wants the bell to be sounded, the window to be flashed, or both.</p>
<pre>Window window_create(owner, type, attributes) Window owner; <window type> type; <attribute-list> attributes;</pre>	<p>Creates a window and returns its handle. type is one of FRAME, PANEL, TEXTSW, TTY, or CANVAS.</p>
<pre>void window_default_event_proc(window, event, arg) Window window; Event *event; caddr_t arg;</pre>	<p>Calls the default event procedure. The arguments passed in are the window (canvas or panel), the event, and an optional argument pertaining to the event.</p>
<pre>window_destroy(win) Window win;</pre>	<p>Destroys win, and any subwindows or subframes owned by win.</p>
<pre>window_done(win) Window win;</pre>	<p>Destroys the entire hierarchy to which win belongs.</p>
<pre>window_fit(win) Window win;</pre>	<p>Causes win to fit its contents in both dimensions. A macro, defined as: <code>window_set(win, WIN_FIT, 0, 0)</code>.</p>
<pre>window_fit_height(win) Window win;</pre>	<p>Causes win to fit its contents in the vertical dimension. A macro, defined as: <code>window_set(win, WIN_FIT_HEIGHT, 0, 0)</code>.</p>
<pre>window_fit_width(win) Window win;</pre>	<p>Causes win to fit its contents in the horizontal dimension. A macro, defined as: <code>window_set(win, WIN_FIT_WIDTH, 0, 0)</code>.</p>
<pre>caddr_t window_get(win, attribute) Window win; Window_attribute attribute;</pre>	<p>Retrieves the value of an attribute for win.</p>

Table 19-36 *Window Functions and Macros—Continued*

<i>Definition</i>	<i>Description</i>
caddr_t window_loop(subframe) Frame subframe;	Causes subframe to be displayed, and receive all input. The call will not return until window_return() is called from one of the application's notify procs.
void window_main_loop(base_frame) Frame base_frame;	Displays base_frame on the screen and begins the processing of events by passing control to the Notifier.
int window_read_event(window, event) Window window; Event *event;	Reads the next input event for window. In case of error, sets the global variable errno and returns -1.
void window_refuse_kbd_focus(window) Window window;	When your event handler receives a KBD_REQUEST event, call this function if you do not want your window to become the keyboard focus.
void window_release_event_lock(window) Window window;	Releases the event lock, allowing other processes to receive input.
void window_return(value) caddr_t value;	Usually called from one of the application's panel item notify procs. Causes window_loop() to return.
window_set(win, attributes) Window win; <attribute-list> attributes;	Sets the value of one or more of win's attributes. attributes is a null-terminated attribute list.

Table 19-37 *Command Line Frame Arguments*

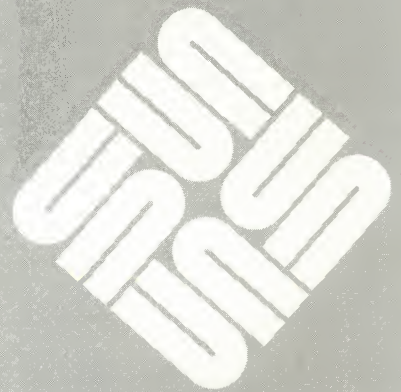
<i>Flag</i>	<i>Long Flag</i>	<i>Arguments</i>	<i>Corresponding Attributes</i>
-Wb	-background_color	<i>red green blue</i>	FRAME_BACKGROUND_COLOR
-Wh	-height	<i>lines</i>	WIN_ROWS
-WH	-help	—	(Causes FRAME_CMDLINE_HELP_PROC to be called.)
-Wf	-foreground_color	<i>red green blue</i>	FRAME_FOREGROUND_COLOR
-Wg	-set_default_color	—	FRAME_INHERIT_COLORS, TRUE
-Wi	-iconic	—	FRAME_CLOSED, TRUE
-WI	-icon_image	<i>filename</i>	ICON_IMAGE of frame's icon ¹⁰⁵
-Wl	-label	<i>label</i>	FRAME_LABEL
-WL	-icon_label	<i>label</i>	ICON_LABEL of frame's icon
-Wn	-no_label	—	FRAME_SHOW_LABEL, FALSE
-Wp	-position	<i>x y</i>	WIN_X, WIN_Y
-WP	-icon_position	<i>x y</i>	FRAME_CLOSED_RECT
-Ws	-size	<i>x y</i>	WIN_WIDTH, WIN_HEIGHT
-Wt	-font	<i>filename</i>	(Sets system default font)
-WT	-icon_font	<i>filename</i>	ICON_FONT of frame's icon
-Ww	-width	<i>columns</i>	WIN_COLUMNS

¹⁰⁵ The -WI option will not work if the application's code does not already specify its icon.

A

Example Programs

Example Programs	389
Source Available	389
A.1. <i>filer</i>	389
A.2. <i>image_browser_1</i>	401
A.3. <i>image_browser_2</i>	406
A.4. <i>tty_io</i>	412
A.5. <i>font_menu</i>	416
A.6. <i>resize_demo</i>	425
A.7. <i>dctool</i>	430
A.8. <i>typein</i>	437
A.9. Programs that Manipulate Color	441
<i>coloredit</i>	441
<i>animatecolor</i>	447
A.10. Two gfx subwindow-based programs converted to use SunView	454
<i>bounce</i>	454
<i>spheres</i>	461



Example Programs

Source Available

If the appropriate optional software category has been installed or mounted on your system, the source code for some of these examples programs is available on-line in `/usr/share/src/sun/suntool/examples`. In addition, the directory above this (`/usr/share/src/sun/suntool`) contains the source for many of the SunView 1 programs in the SunOS, such as `textedit`, `perfmeter`, and `iconedit`.

A.1. *filer*

This program is discussed in Chapter 4, *Using Windows*. It displays a listing in a tty subwindow, which the user manipulates through panel items.

If the user presses the `[Props]` key in the panel, or chooses 'Props' from the frame menu, or pushes the `Set ls flags` button, a pop-up subframe appears. *filer* uses the Selection Service to determine what file name the user has selected, and creates a pop-up text subwindow where that file is displayed.

filer uses the alerts package to ask the user for confirmation and put up messages. It also includes old code which mimics alerts by using `window_loop()` to put up a subframe, but programs written for SunOS Release 4.0 and beyond in general will have no need for this.


```

/*****
/*          4.0      filer.c          */
*****/

#include <suntool/sunview.h>
#include <suntool/panel.h>
#include <suntool/tty.h>
#include <suntool/textsw.h>
#include <suntool/seln.h>
#include <suntool/alert.h>
#include <sys/stat.h>      /* stat call needed to verify existence of files */

/* these objects are global so their attributes can be modified or retrieved */
Frame      base_frame, edit_frame, ls_flags_frame;
Panel      panel, ls_flags_panel;
Tty        ttysw;
Textsw     editsw;
Panel_item dir_item, fname_item, filing_mode_item, done_item;
int        quit_confirmed_from_panel;

#define      MAX_FILENAME_LEN      256
#define      MAX_PATH_LEN          1024

char *getwd();

main(argc, argv)
    int    argc;
    char **argv;
{
    static Notify_value filer_destroy_func();
    void      ls_flags_proc();

    base_frame = window_create(NULL, FRAME,
                                FRAME_ARGS,      argc, argv,
                                FRAME_LABEL,      "filer",
                                FRAME_PROPS_ACTION_PROC, ls_flags_proc,
                                FRAME_PROPS_ACTIVE, TRUE,
                                FRAME_NO_CONFIRM, TRUE,
                                0);
    (void) notify_interpose_destroy_func(base_frame, filer_destroy_func);

    create_panel_subwindow();
    create_tty_subwindow();
    create_edit_popup();
    create_ls_flags_popup();
    quit_confirmed_from_panel = 0;

    window_main_loop(base_frame);
    exit(0);
}

create_tty_subwindow()
{
    ttysw = window_create(base_frame, TTY, 0);
}

create_edit_popup()
{

```

```

edit_frame = window_create(base_frame, FRAME,
                           FRAME_SHOW_LABEL, TRUE,
                           0);
editsw = window_create(edit_frame, TEXTSW, 0);
}

create_panel_subwindow()
{
    void ls_proc(), ls_flags_proc(), quit_proc(), edit_proc(),
        edit_sel_proc(), del_proc();

    char current_dir[MAX_PATH_LEN];

    panel = window_create(base_frame, PANEL, 0);

    (void) panel_create_item(panel, PANEL_BUTTON,
                             PANEL_LABEL_X,          ATTR_COL(0),
                             PANEL_LABEL_Y,          ATTR_ROW(0),
                             PANEL_LABEL_IMAGE,       panel_button_image(panel, "List Directory", 0, 0),
                             PANEL_NOTIFY_PROC,       ls_proc,
                             0);

    (void) panel_create_item(panel, PANEL_BUTTON,
                             PANEL_LABEL_IMAGE,       panel_button_image(panel, "Set ls flags", 0, 0),
                             PANEL_NOTIFY_PROC,       ls_flags_proc,
                             0);

    (void) panel_create_item(panel, PANEL_BUTTON,
                             PANEL_LABEL_IMAGE,       panel_button_image(panel, "Edit", 0, 0),
                             PANEL_NOTIFY_PROC,       edit_proc,
                             0);

    (void) panel_create_item(panel, PANEL_BUTTON,
                             PANEL_LABEL_IMAGE,       panel_button_image(panel, "Delete", 0, 0),
                             PANEL_NOTIFY_PROC,       del_proc,
                             0);

    (void) panel_create_item(panel, PANEL_BUTTON,
                             PANEL_LABEL_IMAGE,       panel_button_image(panel, "Quit", 0, 0),
                             PANEL_NOTIFY_PROC,       quit_proc,
                             0);

    filing_mode_item = panel_create_item(panel, PANEL_CYCLE,
                                         PANEL_LABEL_X,          ATTR_COL(0),
                                         PANEL_LABEL_Y,          ATTR_ROW(1),
                                         PANEL_LABEL_STRING,      "Filing Mode:",
                                         PANEL_CHOICE_STRINGS,     "Use \"File:\" item",
                                         "Use Current Selection", 0,
                                         0);

    (void) panel_create_item(panel, PANEL_MESSAGE,
                             PANEL_LABEL_X,          ATTR_COL(0),
                             PANEL_LABEL_Y,          ATTR_ROW(2),
                             0);

    dir_item = panel_create_item(panel, PANEL_TEXT,
                                  PANEL_LABEL_X,          ATTR_COL(0),
                                  PANEL_LABEL_Y,          ATTR_ROW(3),

```



```

        PANEL_VALUE_DISPLAY_LENGTH,    60,
        PANEL_VALUE,                   getwd(current_dir),
        PANEL_LABEL_STRING,            "Directory: ",
        0);

fname_item = panel_create_item(panel, PANEL_TEXT,
        PANEL_LABEL_X,                 ATTR_COL(0),
        PANEL_LABEL_Y,                 ATTR_ROW(4),
        PANEL_VALUE_DISPLAY_LENGTH,    60,
        PANEL_LABEL_STRING,            "File:  ",
        0);

window_fit_height(panel);

window_set(panel, PANEL_CARET_ITEM, fname_item, 0);
}

create_ls_flags_popup()
{
    void done_proc();
    ls_flags_frame = window_create(base_frame, FRAME, 0);

    ls_flags_panel = window_create(ls_flags_frame, PANEL, 0);

    panel_create_item(ls_flags_panel, PANEL_MESSAGE,
        PANEL_ITEM_X,                   ATTR_COL(14),
        PANEL_ITEM_Y,                   ATTR_ROW(0),
        PANEL_LABEL_STRING,             "Options for ls command",
        PANEL_CLIENT_DATA,              "  ",
        0);

    panel_create_item(ls_flags_panel, PANEL_CYCLE,
        PANEL_ITEM_X,                   ATTR_COL(0),
        PANEL_ITEM_Y,                   ATTR_ROW(1),
        PANEL_DISPLAY_LEVEL,            PANEL_CURRENT,
        PANEL_LABEL_STRING,             "Format:                                ",
        PANEL_CHOICE_STRINGS,           "Short", "Long", 0,
        PANEL_CLIENT_DATA,              " 1 ",
        0);

    panel_create_item(ls_flags_panel, PANEL_CYCLE,
        PANEL_ITEM_X,                   ATTR_COL(0),
        PANEL_ITEM_Y,                   ATTR_ROW(2),
        PANEL_DISPLAY_LEVEL,            PANEL_CURRENT,
        PANEL_LABEL_STRING,             "Sort Order:                            ",
        PANEL_CHOICE_STRINGS,           "Descending", "Ascending", 0,
        PANEL_CLIENT_DATA,              " r ",
        0);

    panel_create_item(ls_flags_panel, PANEL_CYCLE,
        PANEL_ITEM_X,                   ATTR_COL(0),
        PANEL_ITEM_Y,                   ATTR_ROW(3),
        PANEL_DISPLAY_LEVEL,            PANEL_CURRENT,
        PANEL_LABEL_STRING,             "Sort criterion:                        ",
        PANEL_CHOICE_STRINGS,           "Name", "Modification Time",
        PANEL_CLIENT_DATA,              "Access Time", 0,
        0);
    PANEL_CLIENT_DATA,              " tu",
    0);

```

```

panel_create_item(ls_flags_panel, PANEL_CYCLE,
    PANEL_ITEM_X,      ATTR_COL(0),
    PANEL_ITEM_Y,      ATTR_ROW(4),
    PANEL_DISPLAY_LEVEL, PANEL_CURRENT,
    PANEL_LABEL_STRING, "For directories, list:      ",
    PANEL_CHOICE_STRINGS, "Contents", "Name Only", 0,
    PANEL_CLIENT_DATA,  " d ",
    0);

panel_create_item(ls_flags_panel, PANEL_CYCLE,
    PANEL_ITEM_X,      ATTR_COL(0),
    PANEL_ITEM_Y,      ATTR_ROW(5),
    PANEL_DISPLAY_LEVEL, PANEL_CURRENT,
    PANEL_LABEL_STRING, "Recursively list subdirectories? ",
    PANEL_CHOICE_STRINGS, "No", "Yes", 0,
    PANEL_CLIENT_DATA,  " R ",
    0);

panel_create_item(ls_flags_panel, PANEL_CYCLE,
    PANEL_ITEM_X,      ATTR_COL(0),
    PANEL_ITEM_Y,      ATTR_ROW(6),
    PANEL_DISPLAY_LEVEL, PANEL_CURRENT,
    PANEL_LABEL_STRING, "List '.' files?      ",
    PANEL_CHOICE_STRINGS, "No", "Yes", 0,
    PANEL_CLIENT_DATA,  " a ",
    0);

panel_create_item(ls_flags_panel, PANEL_CYCLE,
    PANEL_ITEM_X,      ATTR_COL(0),
    PANEL_ITEM_Y,      ATTR_ROW(6),
    PANEL_DISPLAY_LEVEL, PANEL_CURRENT,
    PANEL_LABEL_STRING, "Indicate type of file?      ",
    PANEL_CHOICE_STRINGS, "No", "Yes", 0,
    PANEL_CLIENT_DATA,  " F ",
    0);

done_item = panel_create_item(ls_flags_panel, PANEL_BUTTON,
    PANEL_ITEM_X,      ATTR_COL(0),
    PANEL_ITEM_Y,      ATTR_ROW(7),
    PANEL_LABEL_IMAGE,  panel_button_image(panel, "Done", 0, 0),
    PANEL_NOTIFY_PROC,  done_proc,
    0);

window_fit(ls_flags_panel); /* fit panel around its items */
window_fit(ls_flags_frame); /* fit frame around its panel */
}

char *
compose_ls_options()
{
    static char  flags[20];
    char        *ptr;
    char        flag;
    int         first_flag = TRUE;
    Panel_item  item;
    char        *client_data;
    int         index;

    ptr = flags;

```



```

panel_each_item(ls_flags_panel, item)
{
    if (item != done_item) {
        client_data = panel_get(item, PANEL_CLIENT_DATA, 0);
        index = (int)panel_get_value(item);
        flag = client_data[index];
        if (flag != ' ') {
            if (first_flag) {
                *ptr++ = '-';
                first_flag = FALSE;
            }
            *ptr++ = flag;
        }
    }
    panel_end_each
    *ptr = '\0';
    return flags;
}

void
ls_proc()
{
    static char previous_dir[MAX_PATH_LEN];
    char *current_dir;
    char cmdstring[100];          /* dir_item's value can be 80, plus flags */

    current_dir = (char *)panel_get_value(dir_item);

    if (strcmp(current_dir, previous_dir)) {
        chdir((char *)panel_get_value(dir_item));
        strcpy(previous_dir, current_dir);
    }
    sprintf(cmdstring, "ls %s %s/%s\n",
            compose_ls_options(),
            current_dir,
            panel_get_value(fname_item));
    ttysw_input(ttysw, cmdstring, strlen(cmdstring));
}

void
ls_flags_proc()
{
    window_set(ls_flags_frame, WIN_SHOW, TRUE, 0);
}

void
done_proc()
{
    window_set(ls_flags_frame, WIN_SHOW, FALSE, 0);
}

/* return a pointer to the current selection */
char *
get_selection()
{
    static char    filename[MAX_FILENAME_LEN];
    Seln_holder    holder;
    Seln_request *buffer;

```

```

    holder = seln_inquire(SELN_PRIMARY);
    buffer = seln_ask(&holder, SELN_REQ_CONTENTS_ASCII, 0, 0);
    strncpy(
        filename, buffer->data + sizeof(Seln_attribute), MAX_FILENAME_LEN);
    return(filename);
}

/* return 1 if file exists, else print error message and return 0 */
stat_file(filename)
    char *filename;
{
    static char previous_dir[MAX_PATH_LEN];
    char *current_dir;
    char this_file[MAX_PATH_LEN];
    struct stat statbuf;

    current_dir = (char *)panel_get_value(dir_item);

    if (strcmp(current_dir, previous_dir)) {
        chdir((char *)panel_get_value(dir_item));
        strcpy(previous_dir, current_dir);
    }
    sprintf(this_file, "%s/%s", current_dir, filename);
    if (stat(this_file, &statbuf) < 0) {
        char buf[MAX_FILENAME_LEN+11]; /* big enough for message */
        sprintf(buf, "%s not found.", this_file);
        msg(buf, 1);
        return 0;
    }
    return 1;
}

void
edit_proc()
{
    void edit_file_proc(), edit_sel_proc();
    int file_mode = (int)panel_get_value(filing_mode_item);

    if (file_mode) {
        (void)edit_sel_proc();
    } else {
        (void)edit_file_proc();
    }
}

void
edit_file_proc()
{
    char *filename;

    /* return if no selection */
    if (!strlen(filename = (char *)panel_get_value(fname_item))) {
        msg("Please enter a value for \"File:\".", 1);
        return;
    }

    /* return if file not found */
    if (!stat_file(filename))

```



```

        return;

        window_set(editsw, TEXTSW_FILE, filename, 0);

        window_set(edit_frame, FRAME_LABEL, filename, WIN_SHOW, TRUE, 0);
    }

void
edit_sel_proc()
{
    char *filename;

    /* return if no selection */
    if (!strlen(filename = get_selection())) {
        msg("Please select a file to edit.", 0);
        return;
    }

    /* return if file not found */
    if (!stat_file(filename))
        return;

    window_set(editsw, TEXTSW_FILE, filename, 0);

    window_set(edit_frame, FRAME_LABEL, filename, WIN_SHOW, TRUE, 0);
}

void
del_proc()
{
    char    buf[300];
    char    *filename;
    int     result;
    Event    event; /* unused */
    int     file_mode = (int)panel_get_value(filing_mode_item);

    /* return if no selection */
    if (file_mode) {
        if (!strlen(filename = get_selection())) {
            msg("Please select a file to delete.", 1);
            return;
        }
    } else {
        if (!strlen(filename = (char *)panel_get_value(fname_item))) {
            msg("Please enter a file name to delete.", 1);
            return;
        }
    }

    /* return if file not found */
    if (!stat_file(filename))
        return;

    /* user must confirm the delete */
    result = alert_prompt(base_frame, &event,
        ALERT_MESSAGE_STRINGS,
        "Ok to delete file:",
        filename,

```

```

        0,
        ALERT_BUTTON_YES,      "Confirm, delete file",
        ALERT_BUTTON_NO,       "Cancel",
    0);
switch (result) {
    case ALERT_YES:
        unlink(filename);
        sprintf(buf, "%s deleted.", filename);
        msg(buf, 0);
        break;
    case ALERT_NO:
        break;
    case ALERT_FAILED: /* not likely to happen unless out of FDs */
        sprintf(buf, "Ok to delete file %s?", filename);
        result = confirm_yes(buf);
        if (result) {
            unlink(filename);
            sprintf(buf, "%s deleted.", filename);
            msg(buf, 1);
        }
        break;
}
}

int
confirm_quit()
{
    int    result;
    Event  event; /* unused */
    char   *msg = "Are you sure you want to Quit?";

    result = alert_prompt(base_frame, &event,
        ALERT_MESSAGE_STRINGS,
        "Are you sure you want to Quit?",
        0,
        ALERT_BUTTON_YES,      "Confirm",
        ALERT_BUTTON_NO,       "Cancel",
        0);
    switch (result) {
        case ALERT_YES:
            break;
        case ALERT_NO:
            return 0;
        case ALERT_FAILED: /* not likely to happen unless out of FDs */
            result = confirm_yes(msg);
            if (!result) {
                return 0;
            }
            break;
    }
    return 1;
}

static Notify_value
filer_destroy_func(client, status)
    Notify_client  client;
    Destroy_status status;
{

```



```

    if (status == DESTROY_CHECKING) {
        if (quit_confirmed_from_panel) {
            return(notify_next_destroy_func(client, status));
        } else if (confirm_quit() == 0) {
            (void) notify_veto_destroy((Notify_client) (LINT_CAST(client)));
            return(NOTIFY_DONE);
        }
    }
    return(notify_next_destroy_func(client, status));
}

void
quit_proc()
{
    if (confirm_quit()) {
        quit_confirmed_from_panel = 1;
        window_destroy(base_frame);
    }
}

msg(msg, beep)
char *msg;
int  beep;
{
    char    buf[300];
    int     result;
    Event   event; /* unused */
    char    *contine_msg = "Press \"Continue\" to proceed.";

    result = alert_prompt(base_frame, &event,
        ALERT_MESSAGE_STRINGS,
        msg,
        contine_msg,
        0,
        ALERT_NO_BEEPING, (beep) ? 0:1,
        ALERT_BUTTON_YES, "Continue",
        ALERT_TRIGGER, ACTION_STOP, /* allow either YES or NO answer */
        0);
    switch (result) {
        case ALERT_YES:
        case ALERT_TRIGGERED: /* result of ACTION_STOP trigger */
            break;
        case ALERT_FAILED: /* not likely to happen unless out of FDs */
            sprintf(buf, "%s Press \"Continue\" to proceed.", msg);
            result = confirm_ok(buf);
            break;
    }
}

/* confirmer routines to be used if alert fails for any reason */

static Frame    init_confirmer();
static int      confirm();
static void      yes_no_ok();

int
confirm_yes(message)
    char    *message;

```

```

{
    return confirm(message, FALSE);
}

int
confirm_ok(message)
    char      *message;
{
    return confirm(message, TRUE);
}

static int
confirm(message, ok_only)
    char      *message;
    int       ok_only;
{
    Frame      confirmer;
    int        answer;

    /* create the confirmer */
    confirmer = init_confirmer(message, ok_only);
    /* make the user answer */
    answer = (int) window_loop(confirmer);
    /* destroy the confirmer */
    window_set(confirmer, FRAME_NO_CONFIRM, TRUE, 0);
    window_destroy(confirmer);
    return answer;
}

static Frame
init_confirmer(message, ok_only)
    char      *message;
    int       ok_only;
{
    Frame      confirmer;
    Panel      panel;
    Panel_item message_item;
    int        left, top, width, height;
    Rect       *r;
    struct pixrect *pr;

    confirmer = window_create(0, FRAME, FRAME_SHOW_LABEL, FALSE, 0);
    panel = window_create(confirmer, PANEL, 0);
    message_item = panel_create_item(panel, PANEL_MESSAGE,
                                     PANEL_LABEL_STRING, message, 0);

    if (ok_only) {
        pr = panel_button_image(panel, "Continue", 8, 0);
        width = pr->pr_width;
    } else {
        pr = panel_button_image(panel, "Cancel", 8, 0);
        width = 2 * pr->pr_width + 10;
    }

    /* center the yes/no or ok buttons under the message */
    r = (Rect *) panel_get(message_item, PANEL_ITEM_RECT);
    left = (r->r_width - width) / 2;
    if (left < 0)

```



```

        left = 0;
        top = rect_bottom(r) + 5;

        if (ok_only) {
            panel_create_item(panel, PANEL_BUTTON,
                PANEL_ITEM_X, left, PANEL_ITEM_Y, top,
                PANEL_LABEL_IMAGE, pr,
                PANEL_CLIENT_DATA, 1,
                PANEL_NOTIFY_PROC, yes_no_ok,
                0);
        } else {
            panel_create_item(panel, PANEL_BUTTON,
                PANEL_ITEM_X, left, PANEL_ITEM_Y, top,
                PANEL_LABEL_IMAGE, pr,
                PANEL_CLIENT_DATA, 0,
                PANEL_NOTIFY_PROC, yes_no_ok,
                0);
            panel_create_item(panel, PANEL_BUTTON,
                PANEL_LABEL_IMAGE, panel_button_image(panel, "Confirm", 8, 0),
                PANEL_CLIENT_DATA, 1,
                PANEL_NOTIFY_PROC, yes_no_ok,
                0);
        }

        window_fit(panel);
        window_fit(confirm);

        /* center the confirmer frame on the screen */
        r = (Rect *) window_get(confirm, WIN_SCREEN_RECT);
        width = (int) window_get(confirm, WIN_WIDTH);
        height = (int) window_get(confirm, WIN_HEIGHT);
        left = (r->r_width - width) / 2;
        top = (r->r_height - height) / 2;
        if (left < 0)
            left = 0;
        if (top < 0)
            top = 0;
        window_set(confirm, WIN_X, left, WIN_Y, top, 0);

        return confirm;
    }

    static void
    yes_no_ok(item, event)
        Panel_item item;
        Event *event;
    {
        window_return(panel_get(item, PANEL_CLIENT_DATA));
    }

```

A.2. *image_browser_1*

The following program is discussed in Chapter 4, *Using Windows*. It lets the user browse through icons and display them. It shows a more complex subwindow layout.


```

/*****
/*                               image_browser_1.c                               */
*****/

#include <suntool/sunview.h>
#include <suntool/panel.h>
#include <suntool/tty.h>
#include <stdio.h>
#include <suntool/icon_load.h>
#include <suntool/seln.h>

Frame frame;
Panel control_panel, display_panel;
Tty  tty;

Panel_item dir_item, fname_item, image_item;

ls_proc(), show_proc(), quit_proc();

char *get_selection();

#define MAX_PATH_LEN 1024
#define MAX_FILENAME_LEN 256

main(argc, argv)
int argc;
char **argv;
{
    frame = window_create(NULL, FRAME,
                          FRAME_ARGS,  argc, argv,
                          FRAME_LABEL, "image_browser_1",
                          0);

    init_tty();
    init_control_panel();
    init_display_panel();
    window_fit(frame);
    window_main_loop(frame);
    exit(0);
}

init_tty()
{
    tty = window_create(frame, TTY,
                        WIN_COLUMNS, 30,
                        WIN_ROWS,    20,
                        0);
}

```

```

init_control_panel()
{
    char *getwd();
    char current_dir[1024];

    control_panel = window_create(frame, PANEL, 0);

    dir_item = panel_create_item(control_panel, PANEL_TEXT,
        PANEL_VALUE_DISPLAY_LENGTH, 13,
        PANEL_LABEL_STRING, "Dir: ",
        PANEL_VALUE, getwd(current_dir),
        0);

    fname_item = panel_create_item(control_panel, PANEL_TEXT,
        PANEL_ITEM_X, ATTR_COL(0),
        PANEL_ITEM_Y, ATTR_ROW(1),
        PANEL_VALUE_DISPLAY_LENGTH, 13,
        PANEL_LABEL_STRING, "File:",
        0);

    panel_create_item(control_panel, PANEL_BUTTON,
        PANEL_ITEM_X, ATTR_COL(0),
        PANEL_ITEM_Y, ATTR_ROW(2),
        PANEL_LABEL_IMAGE, panel_button_image(control_panel, "List", 0, 0),
        PANEL_NOTIFY_PROC, ls_proc,
        0);

    panel_create_item(control_panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(control_panel, "Show", 0, 0),
        PANEL_NOTIFY_PROC, show_proc,
        0);

    panel_create_item(control_panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(control_panel, "Quit", 0, 0),
        PANEL_NOTIFY_PROC, quit_proc,
        0);

    window_fit(control_panel);
}

```



```

ls_proc()
{
    static char previous_dir[MAX_PATH_LEN];
    char *current_dir;
    char cmdstring[100];

    current_dir = (char *)panel_get_value(dir_item);

    if (strcmp(current_dir, previous_dir)) {
        chdir(current_dir);
        sprintf(cmdstring, "cd %s\n", current_dir);
        ttysw_input(tty, cmdstring, strlen(cmdstring));
        strcpy(previous_dir, current_dir);
    }

    sprintf(cmdstring, "ls -l %s\n", panel_get_value(fname_item));
    ttysw_input(tty, cmdstring, strlen(cmdstring));
}

quit_proc()
{
    window_destroy(frame);
}

show_proc()
{
    char *filename;

    if (!strlen(filename = get_selection()))
        return;

    load_image(filename);
}

load_image(filename)
char *filename;
{
    Pixrect *image;
    char error_msg[IL_ERRORMSG_SIZE];

    if (image = icon_load_mpr(filename, error_msg)) {
        panel_set(image_item,
            PANEL_ITEM_X,      ATTR_COL(5),
            PANEL_ITEM_Y,      ATTR_ROW(4),
            PANEL_LABEL_IMAGE, image,
            0);
    }
}

```

```
init_display_panel()
{
    display_panel = window_create(frame, PANEL,
                                WIN_BELOW,    control_panel,
                                WIN_RIGHT_OF, tty,
                                0);
    image_item = panel_create_item(display_panel, PANEL_MESSAGE, 0);
}

char *
get_selection()
{
    static char    filename[MAX_FILENAME_LEN];
    Seln_holder    holder;
    Seln_request    *buffer;

    holder = seln_inquire(SELN_PRIMARY);
    buffer = seln_ask(&holder, SELN_REQ_CONTENTS_ASCII, 0, 0);
    strncpy(filename, buffer->data + sizeof(Seln_attribute), MAX_FILENAME_LEN);
    return (filename);
}
```


A.3. *image_browser_2*

The following program is discussed in Chapter 4, *Using Windows*. It is a more complex icon browser than the previous example. It illustrates how you can use *row/column space* to specify the size of a subwindow.

```

/*****
#ifdef lint
static char sccsid[] = "@(#)image_browser_2.c 1.3 86/09/15 Copyr 1986 Sun Micro";
#endif
*****/

#include <suntool/sunview.h>
#include <suntool/panel.h>
#include <suntool/tty.h>
#include <stdio.h>
#include <suntool/icon_load.h>
#include <suntool/seln.h>
#include <suntool/expand_name.h>
#include <suntool/scrollbar.h>

static char namebuf[100];
static int file_count, image_count;
static struct namelist *name_list;
#define get_name(i) name_list->names[(i)]

Frame frame;
Panel control_panel, display_panel;
Tty tty;

Panel_item dir_item, fname_item, image_item;

show_proc(), browse_proc(), quit_proc();

Pixrect *get_image();

char *get_selection();

#define MAX_PATH_LEN 1024
#define MAX_FILENAME_LEN 256

main(argc, argv)
int argc;
char **argv;
{
    frame = window_create(NULL, FRAME,
                          FRAME_ARGS, argc, argv,
                          FRAME_LABEL, "image_browser_2",
                          0);

    init_control_panel();
    init_display_panel();
    window_set(control_panel,
               WIN_WIDTH, window_get(display_panel, WIN_WIDTH, 0),
               0);
    window_fit(frame);
    window_main_loop(frame);
    exit(0);
}

```



```
init_control_panel()
{
    char current_dir[MAX_PATH_LEN];

    control_panel = window_create(frame, PANEL, 0);

    dir_item = panel_create_item(control_panel, PANEL_TEXT,
        PANEL_LABEL_X,          ATTR_COL(0),
        PANEL_LABEL_Y,          ATTR_ROW(0),
        PANEL_VALUE_DISPLAY_LENGTH, 23,
        PANEL_VALUE,             getwd(current_dir),
        PANEL_LABEL_STRING,       "Dir: ",
        0);

    (void) panel_create_item(control_panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(control_panel, "Browse", 0, 0),
        PANEL_NOTIFY_PROC, browse_proc,
        0);

    fname_item = panel_create_item(control_panel, PANEL_TEXT,
        PANEL_LABEL_X,          ATTR_COL(0),
        PANEL_LABEL_Y,          ATTR_ROW(1),
        PANEL_VALUE_DISPLAY_LENGTH, 23,
        PANEL_LABEL_STRING,       "File:",
        0);

    (void) panel_create_item(control_panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(control_panel, "Quit", 6, 0),
        PANEL_NOTIFY_PROC, quit_proc,
        0);

    window_fit_height(control_panel);

    window_set(control_panel, PANEL_CARET_ITEM, fname_item, 0);
}
```

```

browse_proc()
{
    Panel_item    old_item;
    register int i;
    int          len;
    Pixrect      *image;
    int          previous_image_count;
    register int row, col;

    set_directory();
    match_files();

    panel_each_item(display_panel, old_item)
        pr_destroy ((Pixrect *)panel_get(old_item, PANEL_LABEL_IMAGE));
        panel_free(old_item);
    panel_end_each

    previous_image_count = image_count;
    for (row = 0, image_count = 0; image_count < file_count; row++)
        for (col = 0; col < 4 && image_count < file_count; col++) {
            if (image = get_image(image_count)) {
                panel_create_item(display_panel, PANEL_MESSAGE,
                                PANEL_ITEM_Y,      ATTR_ROW(row),
                                PANEL_ITEM_X,      ATTR_COL(col),
                                PANEL_LABEL_IMAGE, image, 0);
                image_count++;
            }
        }

    if (image_count <= previous_image_count)
        panel_update_scrolling_size(display_panel);

    panel_paint(display_panel, PANEL_CLEAR);

    free_namelist(name_list);
}

set_directory()
{
    static char previous_dir[MAX_PATH_LEN];
    char *current_dir;

    current_dir = (char *)panel_get_value(dir_item);

    if (strcmp(current_dir, previous_dir)) {
        chdir(current_dir);
        strcpy(previous_dir, current_dir);
    }
}

Pixrect *
get_image(i)
int i;
{
    char error_msg[IL_ERRORMSG_SIZE];
    return (icon_load_mpr(get_name(i), error_msg));
}

```



```
match_files()
{
    char *val;

    val = (char *)panel_get_value(fname_item);
    strcpy(namebuf, val);
    name_list = expand_name(namebuf);
    file_count = name_list->count;
}

quit_proc()
{
    window_destroy(frame);
}

show_proc()
{
    char *filename;

    if (!strlen(filename = get_selection()))
        return;

    load_image(filename);
}

load_image(filename)
char *filename;
{
    Pixrect *image;
    char error_msg[IL_ERRORMSG_SIZE];

    if (image = icon_load_mpr(filename, error_msg)) {
        panel_set(image_item,
                  PANEL_ITEM_X,      ATTR_COL(5),
                  PANEL_ITEM_Y,      ATTR_ROW(4),
                  PANEL_LABEL_IMAGE, image,
                  0);
    }
}
```

```

init_display_panel()
{
    int width;
    Scrollbar sb = scrollbar_create(SCROLL_MARGIN,10,0);
    width = (int)scrollbar_get(sb, SCROLL_THICKNESS, 0);
    display_panel = window_create(frame, PANEL,
                                WIN_BELOW,          control_panel,
                                WIN_X,              0,
                                WIN_VERTICAL_SCROLLBAR, sb,
                                WIN_ROW_HEIGHT,     64,
                                WIN_COLUMN_WIDTH,    64,
                                WIN_ROW_GAP,         10,
                                WIN_COLUMN_GAP,      10,
                                WIN_LEFT_MARGIN,     width + 10,
                                WIN_TOP_MARGIN,      10,
                                WIN_ROWS,           4,
                                WIN_COLUMNS,        4,
                                0);
    window_set(display_panel, WIN_LEFT_MARGIN, 10, 0);
}

char *
get_selection()
{
    static char    filename[MAX_FILENAME_LEN];
    Seln_holder    holder;
    Seln_request   *buffer;

    holder = seln_inquire(SELN_PRIMARY);
    buffer = seln_ask(&holder, SELN_REQ_CONTENTS_ASCII, 0, 0);
    strncpy(filename, buffer->data + sizeof(Seln_attribute), MAX_FILENAME_LEN);
    return (filename);
}

```


A.4. *tty_io*

The following program demonstrates the use of `ttysw_input()`, `ttysw_output()` and TTY escape sequences. These functions are explained in Chapter 11, *TTY Subwindows*.

tty_io creates a panel and a tty subwindow. You can send arbitrary character sequences to the latter as input or output by manipulating panel items. There is also a button that sends the current time within the escape sequence to set the frame label. Try sending different sequences to the tty subwindow. Press CTRL-R to see the difference between what appears on the screen and what was input to the pseudo-tty. Also try starting the tool with a program such as *vi* as a command line argument.

```

/*****
#include <stdio.h>
static char sccsid[] = "@(#)tty_io.c 1.4 87/11/19 Copyr 1986 Sun Micro";
#endif
*****/

#include <stdio.h>
#include <suntool/sunview.h>
#include <suntool/tty.h>
#include <suntool/panel.h>

#define TEXT_ITEM_MAX_LENGTH 25

Tty          tty;
Panel_item   text_item;
char         tmp_buf[80];

static void   input_text();
static void   output_text();
static void   output_time();

main(argc, argv)
    int      argc;
    char     **argv;
{
    Frame     frame;
    Panel     panel;

    frame = window_create(NULL, FRAME,
                          FRAME_ARGS,      argc, argv,
                          WIN_ERROR_MSG,   "Can't create tool frame",
                          0);
    panel = window_create(frame, PANEL, 0);

    /* set up a simple panel subwindow */
    panel_create_item(panel, PANEL_BUTTON,
                      PANEL_LABEL_IMAGE, panel_button_image(panel, "Input text", 11, 0),
                      PANEL_NOTIFY_PROC, input_text,
                      0);
    panel_create_item(panel, PANEL_BUTTON,
                      PANEL_LABEL_IMAGE, panel_button_image(panel, "Output text", 11, 0),
                      PANEL_NOTIFY_PROC, output_text,
                      0);
    text_item = panel_create_item(panel, PANEL_TEXT,
                                  PANEL_LABEL_STRING, "Text:",
                                  PANEL_VALUE, "Hello hello",
                                  PANEL_VALUE_DISPLAY_LENGTH, TEXT_ITEM_MAX_LENGTH,
                                  0);
    panel_create_item(panel, PANEL_BUTTON,
                      PANEL_LABEL_IMAGE, panel_button_image(panel, "Show time", 11, 0),
                      PANEL_NOTIFY_PROC, output_time,
                      0);

    window_fit_height(panel);

```



```
/* Assume rest of arguments are for tty subwindow, except FRAME_ARGS leaves the
 * program_name as argv[0], and we don't want to pass this to the tty subwindow.
 */
argv++;
tty = window_create(frame, TTY,
                    TTY_ARGV,      argv,
                    WIN_ROWS,      24,
                    WIN_COLUMNS,   80,
                    0);

window_fit(frame);

ttysw_input(tty, "echo my pseudo-tty is `tty`\n", 28);

window_main_loop(frame);
exit(0);
}

static void
input_text(item, event)
    Panel_item    item;
    Event         *event;
{
    strcpy(tmp_buf, (char *) panel_get_value(text_item));
    ttysw_input(tty, tmp_buf, strlen(tmp_buf));
}

static void
output_text(item, event)
    Panel_item    item;
    Event         *event;
{
    strcpy(tmp_buf, (char *) panel_get_value(text_item));
    ttysw_output(tty, tmp_buf, strlen(tmp_buf));
}
```

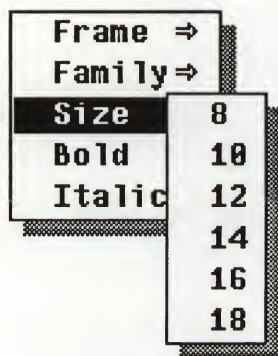
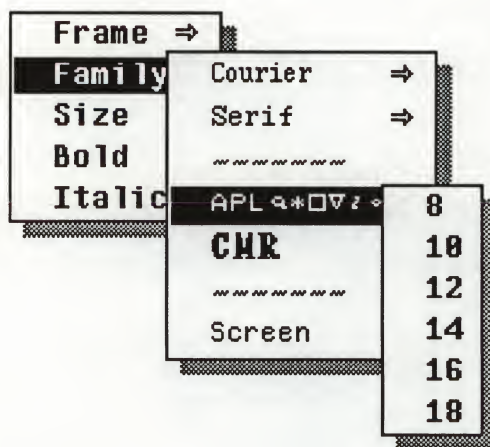
```
static void
output_time(item, event)
    Panel_item    item;
    Event         *event;
{
#include <sys/time.h>
#define ASCTIMELEN    26

    struct timeval  tp;

    /* construct escape sequence to set frame label */
    tmp_buf[0] = '\033';
    tmp_buf[1] = ']';
    tmp_buf[2] = '1';
    tmp_buf[2 + ASCTIMELEN + 1] = '\033';
    tmp_buf[2 + ASCTIMELEN + 2] = '\\';
    gettimeofday(&tp, NULL);
    strncpy(&tmp_buf[3], ctime(&tp.tv_sec), ASCTIMELEN);
    ttysw_output(tty, tmp_buf, ASCTIMELEN + 5);
}
```


A.5. *font_menu*

The next program, *font_menu*, builds on several of the examples given in Chapter 12, *Menus*. Examples of the font menu it creates are shown below:



```

/*****
#define lint
static char sccsid[] = "@(#)font_menu.c 1.2 86/09/15 Copyr 1986 Sun Micro";
#endif
*****/

#include <suntool/sunview.h>
#include <suntool/panel.h>
#include <suntool/walkmenu.h>

void set_family(), set_size(), set_on_off(), toggle_on_off(), open_fonts();
Menu new_menu(), initialize_on_off();
char *int_to_str();
extern char * sprintf();
extern char * malloc();

Panel_item feedback_item;
char *family, *size, *bold, *italic;
Pixfont *cour, *serif, *apl, *cmr, *screen;

/*****
/* main
/* First create the base frame, the feedback panel and feedback item. The
/* feedback item is initialized to "gallant 8".
/* Then get the frame's menu, call new_menu() to create a new menu with the
/* original frame menu as a pullright, and give the new menu to the frame.
*****/

main(argc, argv)
    int argc;
    char *argv[];
{
    Frame frame;
    Panel panel;
    Menu menu;
    int defaults;

    frame = window_create(NULL, FRAME, FRAME_LABEL, "Menu Test -- Try frame menu.", 0);
    panel = window_create(frame, PANEL, WIN_ROWS, 1, 0);
    feedback_item = panel_create_item(panel, PANEL_MESSAGE, PANEL_LABEL_STRING, "", 0);

    family = "Gallant", size = "8", bold = italic = "";
    update_feedback();

    /* remember if user gave -d flag */
    if (argc >= 2) defaults = strcmp(argv[1], "-d") == 0;

    menu = (Menu>window_get(frame, WIN_MENU);
    menu = new_menu(menu, defaults);
    window_set(frame, WIN_MENU, menu, 0);

    window_main_loop(frame);
}

```



```

/*****
/* new_menu -- returns a new menu with 'original menu' as a pullright. */
*****/

Menu
new_menu(original_menu, defaults)
    Menu original_menu;
    int defaults;
{
    Menu new_menu, family_menu, size_menu, on_off_menu;
    int i;

    /* create the on-off menu, which will be used as a pullright
     * for both the bold and italic items to the new menu.
     */
    on_off_menu = menu_create(MENU_STRING_ITEM, "On", 1,
                             MENU_STRING_ITEM, "Off", 0,
                             MENU_GEN_PROC, initialize_on_off,
                             MENU_NOTIFY_PROC, set_on_off,
                             0);

    /* create the new menu which will eventually be returned */

    open_fonts(); /* first open the needed fonts */
    new_menu = menu_create(
        MENU_PULLRIGHT_ITEM,
        "Frame",
        original_menu,
        MENU_PULLRIGHT_ITEM,
        "Family",
        family_menu = menu_create(
            MENU_ITEM,
            MENU_STRING, "Courier",
            MENU_FONT, cour,
            0,
            MENU_ITEM,
            MENU_STRING, "Serif",
            MENU_FONT, serif,
            0,
            MENU_ITEM,
            MENU_STRING, "aplAPLGIJ",
            MENU_FONT, apl,
            0,
            MENU_ITEM,
            MENU_STRING, "CMR",
            MENU_FONT, cmr,
            0,

```

```

        MENU_ITEM,
        MENU_STRING, "Screen",
        MENU_FONT,   screen,
        0,
        MENU_NOTIFY_PROC, set_family,
    0),
MENU_PULLRIGHT_ITEM,
    "Size", size_menu = menu_create(0),
MENU_ITEM,
    MENU_STRING,      "Bold",
    MENU_PULLRIGHT,   on_off_menu,
    MENU_NOTIFY_PROC, toggle_on_off,
    MENU_CLIENT_DATA, &bold,
    0,
MENU_ITEM,
    MENU_STRING,      "Italic",
    MENU_PULLRIGHT,   on_off_menu,
    MENU_NOTIFY_PROC, toggle_on_off,
    MENU_CLIENT_DATA, &italic,
    0,
0);

/* give each item in the family menu the size menu as a pullright */
for (i = (int)menu_get(family_menu, MENU_NITEMS); i > 0; --i)
    menu_set(menu_get(family_menu, MENU_NTH_ITEM, i),
        MENU_PULLRIGHT, size_menu, 0);

/* put non-selectable lines inbetween groups of items in family menu */
menu_set(family_menu,
    MENU_INSERT, 2, menu_create_item(MENU_STRING,      "-----",
                                     MENU_INACTIVE,    TRUE,
                                     0),
    0);
menu_set(family_menu,
    MENU_INSERT, 5, menu_get(family_menu, MENU_NTH_ITEM, 3),
    0);

/* The size menu was created with no items. Now give it items representing */
/* the point sizes 8, 10, 12, 14, 16, and 18. */
for (i = 8; i <= 18; i += 2)
    menu_set(size_menu, MENU_STRING_ITEM, int_to_str(i), i, 0);

/* give the size menu a notify proc to update the feedback */
menu_set(size_menu, MENU_NOTIFY_PROC, set_size, 0);

```



```
/* if the user did not give the -d flag, make all the menus come
 * up with the initial and default selections the last selected
 * item, and the initial selection selected.
 */
if (!defaults) {
    menu_set(new_menu,
        MENU_DEFAULT_SELECTION, MENU_SELECTED,
        MENU_INITIAL_SELECTION, MENU_SELECTED,
        MENU_INITIAL_SELECTION_SELECTED, TRUE,
        0);
    menu_set(family_menu,
        MENU_DEFAULT_SELECTION, MENU_SELECTED,
        MENU_INITIAL_SELECTION, MENU_SELECTED,
        MENU_INITIAL_SELECTION_SELECTED, TRUE,
        0);
    menu_set(size_menu,
        MENU_DEFAULT_SELECTION, MENU_SELECTED,
        MENU_INITIAL_SELECTION, MENU_SELECTED,
        MENU_INITIAL_SELECTION_SELECTED, TRUE,
        0);
    menu_set(on_off_menu,
        MENU_DEFAULT_SELECTION, MENU_SELECTED,
        MENU_INITIAL_SELECTION, MENU_SELECTED,
        MENU_INITIAL_SELECTION_SELECTED, TRUE,
        0);
}

return (new_menu);
}
```

```

/*****
/* set_family -- notify proc for family menu.  Get the current family and */
/* display it in the feedback panel.  Note that we first get the value */
/* of the menu item.  This has the side effect of causing any pullrights */
/* further to the right of mi to be evaluated.  Specifically, the value of */
/* each family item is the value of its pullright -- namely the size menu. */
/* When the size menu is evaluated, the notify proc set_size() is called, */
/* which updates the feedback for the size. */
*****/

/*ARGSUSED*/
void
set_family(m, mi)
    Menu m;
    Menu_item mi;
{
    menu_get(mi, MENU_VALUE); /* force pullrights to be evaluated */
    family = menu_get(mi, MENU_STRING);
    update_feedback();
}

/*****
/* set_size -- notify proc for the size menu. */
*****/

/*ARGSUSED*/
void
set_size(m, mi)
    Menu m;
    Menu_item mi;
{
    size = menu_get(mi, MENU_STRING);
    update_feedback();
}

```



```

/*****
/* initialize_on_off -- generate proc for the on_off menu.          */
/* The on-off menu is a pullright of both the bold and the italic menus. */
/* We want it to toggle -- if its parent was on, it should come up with */
/* "Off" selected, and vice-versa. We can do that by first getting the */
/* parent menu item, then, indirectly through its client data attribute, */
/* seeing if the string representing the bold or italic state is null.    */
/* If the string was null, we set the first item ("On") to be selected,  */
/* else we set the second item ("Off") to be selected.                  */
*****/

```

Menu

```

initialize_on_off(m, op)
    Menu m; Menu_generate op;
{
    Menu_item parent_mi;
    char **name;

    if (op != MENU_CREATE) return (m);

    parent_mi = (Menu_item)menu_get(m, MENU_PARENT);
    name = (char **)menu_get(parent_mi, MENU_CLIENT_DATA);

    if (**name == NULL)
        menu_set(m, MENU_SELECTED, 1, 0);
    else
        menu_set(m, MENU_SELECTED, 2, 0);
    return (m);
}

```

```

/*****
/* set_on_off -- notify proc for on-off menu.                               */
/* Set the feedback string -- italic or bold -- appropriately depending on  */
/* the current setting. Note that the "On" item was created to return a      */
/* value of 1, and the "Off" item will return a value of 0.                  */
*****/

void
set_on_off(m, mi)
    Menu m; Menu_item mi;
{
    Menu_item parent_mi;
    char **name;

    parent_mi = (Menu_item)menu_get(m, MENU_PARENT);
    name = (char **)menu_get(parent_mi, MENU_CLIENT_DATA);
    if (menu_get(mi, MENU_VALUE))
        *name = (char *)menu_get(parent_mi, MENU_STRING);
    else
        *name = "";
    update_feedback();
}

/*****
/* toggle_on_off -- notify proc for the "Bold" and "Italic" menu items.      */
/* Using a notify proc for the menu item allows toggling without bringing   */
/* up the on-off pullright.                                                  */
*****/

/*ARGSUSED*/
void
toggle_on_off(m, mi)
    Menu m;
    Menu_item mi;
{
    char **name;

    name = (char **)menu_get(mi, MENU_CLIENT_DATA);

    if (**name == NULL)
        *name = (char *)menu_get(mi, MENU_STRING);
    else
        *name = "";

    update_feedback();
}

```



```
update_feedback()
{
    char buf[30];

    sprintf(buf, "%s %s %s %s", bold, italic, family, size);
    panel_set(feedback_item, PANEL_LABEL_STRING, buf, 0);
}

char *
int_to_str(n)
{
    char *r = malloc(4);
    sprintf(r, "%d", n);
    return (r);
}

void
open_fonts()
{
    cour = pf_open("/usr/lib/fonts/fixedwidthfonts/cour.r.10");
    serif = pf_open("/usr/lib/fonts/fixedwidthfonts/serif.r.10");
    apl = pf_open("/usr/lib/fonts/fixedwidthfonts/apl.r.10");
    cmr = pf_open("/usr/lib/fonts/fixedwidthfonts/cmr.b.8");
    screen = pf_open("/usr/lib/fonts/fixedwidthfonts/screen.r.11");
}
```

A.6. *resize_demo*

This program demonstrates how to resize the subwindows of a frame yourself if you need to use a complicated topology.

The particular subwindow layout used here has four subwindows. One has a fixed width and height in pixels, another has a fixed width in characters (using the user-set default font), and the other two fill up the empty space. One of the subwindows also has a scrollbar.

This program interposes in front of the frame's client event handler. If the event is `WIN_RESIZE`, the program's own `resize()` procedure is called, which sets the subwindow positions explicitly.

For a discussion of interposing and the Notifier, see Chapter 17, *The Notifier*. The simpler case of using window attributes to layout subwindows is described under *Explicit Subwindow Layout* in Chapter @NumberOf(window), @TitleOf(window).


```

/*****
#endif lint
static char sccsid[] = "%Z%M% %I% %E% Copyr 1986 Sun Micro";
#endif
*****/

#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <suntool/scrollbar.h>

Canvas Canvas_1, Canvas_2, Canvas_3, Canvas_4;
Pixwin *Pixwin_1, *Pixwin_2, *Pixwin_3, *Pixwin_4;
Rect framerect;
PIXFONT *font;

extern char * sprintf();
/*
 * font macros:
 *     font_offset(font) gives the vertical distance between
 *                       the font origin and the top left corner
 *                       of the bounding box of the string displayed
 *                       (see Text Facilities for Pixrects in the
 *                       Pixrect Reference Manual)
 *     font_height(font) gives the height of the font
 */

#define font_offset(font)      (-font->pf_char['n'].pc_home.y)
#define font_height(font)     (font->pf_defaultsize.y)

/*
 * SunView-dependent size definitions
 */

#define LEFT_MARGIN      5           /* margin on left side of frame */
#define RIGHT_MARGIN     5           /* margin on right side of frame */
#define BOTTOM_MARGIN     5           /* margin on bottom of frame */
#define SUBWINDOW_SPACING 5         /* space in between adjacent
                                     subwindows */

/*
 * application-dependent size definitions
 */

#define CANVAS_1_WIDTH    320        /* width in pixels of canvas 1 */
#define CANVAS_1_HEIGHT  160        /* height in pixels of canvas 1 */
#define CANVAS_3_COLUMNS  30         /* width in characters of canvas 3 */

main(argc, argv)
int argc;
char **argv;
{
    Frame frame;
    static Notify_value catch_resize();
    static void draw_canvas_1(), draw_canvas_3();

    /*
     * create the frame and subwindows, and open the font

```

```

    * no size attributes are given yet
    */

    frame = window_create(NULL, FRAME,
                          FRAME_ARGS, argc, argv,
                          WIN_ERROR_MSG, "Can't create tool frame",
                          FRAME_LABEL, "Resize Demo",
                          0);
    Canvas_1 = window_create(frame, CANVAS,
                          CANVAS_RESIZE_PROC, draw_canvas_1,
                          0);
    Canvas_2 = window_create(frame, CANVAS,
                          0);
    Canvas_3 = window_create(frame, CANVAS,
                          WIN_VERTICAL_SCROLLBAR, scrollbar_create(
                          SCROLL_PLACEMENT, SCROLL_EAST,
                          0),
                          CANVAS_RESIZE_PROC, draw_canvas_3,
                          0);
    Canvas_4 = window_create(frame, CANVAS,
                          0);
    Pixwin_1 = canvas_pixwin(Canvas_1);
    Pixwin_2 = canvas_pixwin(Canvas_2);
    Pixwin_3 = canvas_pixwin(Canvas_3);
    Pixwin_4 = canvas_pixwin(Canvas_4);
    font = pf_default();

    /*
    * now that the frame and font sizes are known, set the initial
    * subwindow sizes
    */

    resize(frame);

    /*
    * insert an interposer so that whenever the window changes
    * size we will know about it and handle it ourselves
    */

    (void) notify_interpose_event_func(frame, catch_resize, NOTIFY_SAFE);

    /*
    * start execution
    */

    window_main_loop(frame);
    exit(0);
}

/*
* catch_resize
*
* interposed function which checks all input events passed to the frame
* for resize events; if it finds one, resize() is called to refit
* the subwindows; checking is done AFTER the frame processes the
* event because if the frame changes its size due to this event (because
* the window has been opened or closed for instance) we want to fit
* the subwindows to the new size

```



```

*/

static Notify_value
catch_resize(frame, event, arg, type)
    Frame frame;
    Event *event;
    Notify_arg arg;
    Notify_event_type type;
{
    Notify_value value;

    value = notify_next_event_func(frame, event, arg, type);
    if (event_action(event) == WIN_RESIZE)
        resize(frame);
    return(value);
}

/*
 * resize
 *
 * fit the subwindows of the frame to its new size
 */

resize(frame)
    Frame frame;
{
    Rect *r;
    int canvas_3_width;      /* the width in pixels of canvas 3 */
    int stripeheight;        /* the height of the frame's name stripe */

    /* if the window is iconic, don't do anything */

    if ((int>window_get(frame, FRAME_CLOSED))
        return;

    /* find out our new size parameters */

    r = (Rect *) window_get(frame, WIN_RECT);
    framerect = *r;
    stripeheight = (int) window_get(frame, WIN_TOP_MARGIN);

    canvas_3_width = CANVAS_3_COLUMNS * font->pf_defaultsizex
        + (int) scrollbar_get(SCROLLBAR, SCROLL_THICKNESS);
    window_set(Canvas_2,
        WIN_X,          0,
        WIN_Y,          0,
        WIN_WIDTH,      framerect.r_width - canvas_3_width
                        - LEFT_MARGIN - SUBWINDOW_SPACING
                        - RIGHT_MARGIN,
        WIN_HEIGHT,     framerect.r_height - CANVAS_1_HEIGHT
                        - stripeheight - SUBWINDOW_SPACING -
                        BOTTOM_MARGIN,
        0);
    window_set(Canvas_1,
        WIN_X,          0,
        WIN_Y,          framerect.r_height - CANVAS_1_HEIGHT -
                        SUBWINDOW_SPACING - stripeheight,
        WIN_WIDTH,      CANVAS_1_WIDTH,

```

```

        WIN_HEIGHT,    CANVAS_1_HEIGHT,
        0);
window_set(Canvas_4,
        WIN_X,          CANVAS_1_WIDTH + SUBWINDOW_SPACING,
        WIN_Y,          framerect.r_height - CANVAS_1_HEIGHT
                        - SUBWINDOW_SPACING - stripeheight,
        WIN_WIDTH,      framerect.r_width - canvas_3_width
                        - CANVAS_1_WIDTH - LEFT_MARGIN
                        - 2 * SUBWINDOW_SPACING - RIGHT_MARGIN,
        WIN_HEIGHT,    CANVAS_1_HEIGHT,
        0);
window_set(Canvas_3,
        WIN_X,          framerect.r_width - canvas_3_width
                        - LEFT_MARGIN - SUBWINDOW_SPACING,
        WIN_Y,          0,
        WIN_WIDTH,      canvas_3_width,
        WIN_HEIGHT,    framerect.r_height - stripeheight
                        - BOTTOM_MARGIN,
        0);
}

/*
 * draw_canvas_1
 * draw_canvas_3
 *
 * draw simple messages in the canvases
 */

static void
draw_canvas_1()
{
    char buf[64];

    sprintf(buf, "%d by %d pixels",
            CANVAS_1_WIDTH, CANVAS_1_HEIGHT);
    pw_text(Pixwin_1, 5, font_offset(font), PIX_SRC, font,
            "This subwindow is always ");
    pw_text(Pixwin_1, 5, font_offset(font) + font_height(font),
            PIX_SRC, font, buf);
}

static void
draw_canvas_3()
{
    char buf[64];

    sprintf(buf, "%d characters wide",
            CANVAS_3_COLUMNS);
    pw_text(Pixwin_3, 5, font_offset(font), PIX_SRC, font,
            "This subwindow is always ");
    pw_text(Pixwin_3, 5, font_offset(font) + font_height(font), PIX_SRC,
            font, buf);
}

```


A.7. *dctool*

dctool is a simple reverse-polish notation calculator which demonstrates how to use pipes to write a SunView-based front end for an existing non-SunView program. *dctool* consists of a panel with buttons for each digit, the four arithmetic operations, and an enter key. The digits you hit are displayed in a message item and are sent via a pipe to `dc(1)` a UNIX desk calculator. When `dc` computes an answer, it is sent back to *dctool* via a second pipe and it is displayed.

Note also the use of a single notify procedure for all of the digit buttons. The actual digit associated with each button is stored as the client data for each panel item, so the notify procedure can determine which button was pressed by looking at the client data. This value is then passed directly to `dc`. The operation buttons also all use a single notify procedure.

When you run *dctool*, remember that it is a reverse-polish notation calculator. For instance, to compute $3 * 5$ you must hit the buttons 3, Enter, 5, and * in that order. If you prefer infix notation, you could easily adapt *dctool* to use `bc(1)` instead of `dc`.

```

/*****
#endif lint
static char sccsid[] = "@(#)dctool.c 1.4 86/09/15 Copyr 1986 Sun Micro";
#endif
*****/

#include <stdio.h>
#include <suntool/sunview.h>
#include <suntool/panel.h>

static Frame    frame;
static Panel    panel;
static Panel_item digit_item[10], enter_item;
static Panel_item add_item, sub_item, mul_item, div_item;
static Panel_item display_item;

static char      display_buf[512] = ""; /* storage for the
                                         * numbers currently on
                                         * the display (stored as
                                         * a string) */

static FILE      *fp_tochild; /* fp of pipe to child (write
                               * data on it) */
static FILE      *fp_fromchild; /* fp of pipe from child (read
                                  * data from it) */
static int        tochild; /* associated file descriptors */
static int        fromchild;

static int        childpid; /* pid of child process */

static int        dead = 0; /* set to 1 if child process has
                             * died */

main(argc, argv)
    int    argc;
    char   **argv;
{
    static Notify_value pipe_reader();
    static Notify_value dead_child();

    frame = window_create(NULL, FRAME,
                          FRAME_ARGS, argc, argv,
                          WIN_ERROR_MSG, "Cannot create frame",
                          FRAME_LABEL, "dctool - RPN Calculator",
                          0);

    panel = window_create(frame, PANEL,
                          0);
    create_panel_items(panel);

    window_fit(panel);
    window_fit(frame);

    /* start the child process and tell the notifier about it */
    start_dc();
    /*
     * note that notify_set_input_func takes a file descriptor,
     * not a file pointer used by the standard I/O library

```



```

    */
    (void) notify_set_input_func(frame, pipe_reader, fromchild);
    (void) notify_set_wait3_func(frame, dead_child, childpid);

    window_main_loop(frame);
    exit(0);
}

static
create_panel_items(panel)
    Panel          panel;
{
    int            c;
    char           name[2];
    static void     digit_proc(), op_proc();
    static struct {
        int         col, row;
    }              positions[10] = {
        { 0, 3 }, { 0, 0 }, { 6, 0 }, { 12, 0 },
        { 0, 1 }, { 6, 1 }, { 12, 1 },
        { 0, 2 }, { 6, 2 }, { 12, 2 }
    };

    name[1] = '\0';
    for (c = 0; c < 10; c++) {
        name[0] = c + '0';
        digit_item[c] = panel_create_item(panel, PANEL_BUTTON,
            PANEL_LABEL_IMAGE, panel_button_image(panel, name, 3, 0),
            PANEL_NOTIFY_PROC, digit_proc,
            PANEL_CLIENT_DATA, (caddr_t) (c + '0'),
            PANEL_LABEL_X,     ATTR_COL(positions[c].col),
            PANEL_LABEL_Y,     ATTR_ROW(positions[c].row),
            0);
    }
    add_item = panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel, "+", 3, 0),
        PANEL_NOTIFY_PROC, op_proc,
        PANEL_CLIENT_DATA, (caddr_t) '+',
        PANEL_LABEL_X,     ATTR_COL(18),
        PANEL_LABEL_Y,     ATTR_ROW(0),
        0);
    sub_item = panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel, "-", 3, 0),
        PANEL_NOTIFY_PROC, op_proc,
        PANEL_CLIENT_DATA, (caddr_t) '-',
        PANEL_LABEL_X,     ATTR_COL(18),
        PANEL_LABEL_Y,     ATTR_ROW(1),
        0);
    mul_item = panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel, "*", 3, 0),
        PANEL_NOTIFY_PROC, op_proc,
        PANEL_CLIENT_DATA, (caddr_t) '*',
        PANEL_LABEL_X,     ATTR_COL(18),
        PANEL_LABEL_Y,     ATTR_ROW(2),
        0);
    div_item = panel_create_item(panel, PANEL_BUTTON,
        PANEL_LABEL_IMAGE, panel_button_image(panel, "/", 3, 0),
        PANEL_NOTIFY_PROC, op_proc,

```

```

    PANEL_CLIENT_DATA, (caddr_t) '/',
    PANEL_LABEL_X,      ATTR_COL(18),
    PANEL_LABEL_Y,      ATTR_ROW(3),
    0);
enter_item = panel_create_item(panel, PANEL_BUTTON,
    PANEL_LABEL_IMAGE, panel_button_image(panel, "Enter", 7, 0),
    PANEL_NOTIFY_PROC, op_proc,
    PANEL_CLIENT_DATA, (caddr_t) ' ',
    PANEL_LABEL_X,      ATTR_COL(6),
    PANEL_LABEL_Y,      ATTR_ROW(3),
    0);
display_item = panel_create_item(panel, PANEL_MESSAGE,
    PANEL_LABEL_STRING, "0",
    PANEL_LABEL_X,      ATTR_COL(0),
    PANEL_LABEL_Y,      ATTR_ROW(4),
    0);
}

/* callback procedure called whenever a digit button is pressed */

static void
digit_proc(item, event)
    Panel_item    item;
    Event         *event;
{
    int           digit_name = (int) panel_get(item,
                                                PANEL_CLIENT_DATA);
    char          buf[2];

    buf[0] = digit_name;      /* display digit */
    buf[1] = '\0';
    strcat(display_buf, buf);
    panel_set(display_item, PANEL_LABEL_STRING, display_buf, 0);
    send_to_dc(digit_name);   /* send digit to dc */
}

/*
 * callback procedure called whenever an operation button is
 * pressed
 */

static void
op_proc(item, event)
    Panel_item    item;
    Event         *event;
{
    int           op_name = (int) panel_get(item,
                                                PANEL_CLIENT_DATA);

    display_buf[0] = '\0';    /* don't erase display yet; wait
                               * until the answer comes back */
    send_to_dc(op_name);
    if (item != enter_item)
        send_to_dc('p');      /* send a p so the answer will be
                               * printed by dc */
    send_to_dc('\n');
}

```



```

/*
 * start the child process
 */

static
start_dc()
{
    int          pipeto[2], pipefrom[2];
    int          c, numfds;

    if (pipe(pipeto) < 0 || pipe(pipefrom) < 0) {
        perror("dctool");
        exit(1);
    }
    switch (childpid = fork()) {

    case -1:
        perror("dctool");
        exit(1);

    case 0:
        /* this is the child process */
        /*
         * use dup2 to set the child's stdin and stdout to the
         * pipes
         */
        dup2(pipeto[0], 0);
        dup2(pipefrom[1], 1);

        /*
         * close all other fds (except stderr) since the child
         * process doesn't know about or need them
         */

        numfds = getdtablesize();
        for (c = 3; c < numfds; c++)
            close(c);

        /* exec the child process */

        execl("/usr/bin/dc", "dc", 0);
        perror("dctool (child)");      /* shouldn't get here */
        exit(1);

    default:
        /* this is the parent */
        close(pipeto[0]);
        close(pipefrom[1]);
        tochild = pipeto[1];
        fp_tochild = fdopen(tochild, "w");
        fromchild = pipefrom[0];
        fp_fromchild = fdopen(fromchild, "r");

        /*
         * the pipe to dc must be unbuffered or dc will not get
         * any data until 1024 characters have been sent
         */

        setbuf(fp_tochild, NULL);

```

```

        break;
    }
}

/*
 * notify proc called whenever there is data to read on the pipe
 * from the child process; in this case it is an answer from dc,
 * so we display it
 */

static      Notify_value
pipe_reader(frame, fd)
    Frame      frame;
    int        fd;
{
    char        buf[512];

    fgets(buf, 512, fp_fromchild);
    buf[strlen(buf) - 1] = '\0'; /* remove newline */
    panel_set(display_item, PANEL_LABEL_STRING, buf, 0);
    display_buf[0] = '\0';
    return (NOTIFY_DONE);
}

/*
 * notify proc called if the child dies
 */

static      Notify_value
dead_child(frame, pid, status, rusage)
    Frame      frame;
    int        pid;
    union wait  *status;
    struct rusage *rusage;
{
    panel_set(display_item, PANEL_LABEL_STRING, "Child died!", 0);
    dead = 1;

    /*
     * tell the notifier to stop reading the pipe (since it is
     * invalid now)
     */

    (void) notify_set_input_func(frame, NOTIFY_FUNC_NULL,
                                fromchild);

    close(tochild);
    close(fromchild);
    return (NOTIFY_DONE);
}

/* send a character over the pipe to dc */

static
send_to_dc(c)
    char        c;
{
    if (dead)
        panel_set(display_item,

```



```
                PANEL_LABEL_STRING, "Child is dead!",  
                0);  
    else  
        putc(c, fp_tochild);  
}
```

A.8. *typein*

This program shows how to replace the functionality of the Graphics Tool and gfxsw package previously available under SunWindows. *typein* provides a tty emulator for interaction with the user and a canvas to draw on. To demonstrate it, a simple application is included which allows the user to input coordinates in the tty emulator and then draws the vectors in the canvas.

typein uses a tty subwindow and a canvas. Normally, the tty subwindow is used to allow a child process to run in a window; in this case, we would like the same process to write in that window. To accomplish this, we tell the tty subwindow not to fork a child process with the `TTY_ARGV_DO_NOT_FORK` value for `TTY_ARGV`. *typein* uses `dup2(2)` to set its stdin and stdout to the `TTY_FD`. When the user types something in the tty subwindow, *typein*'s `read_input()` routine is called.

NOTE When using this mechanism, be careful of the following problems. First, you must use the Notifier (unlike the old gfxsw). Second, if you use the standard I/O package, be sure to either use `fflush` carefully or to remove all buffering with `setbuf` because the package will think you are sending data to a file and not to a tty. Finally, be sure you never block on a read because the program will hang (either use non-blocking I/O or only read one line at a time).


```

/*****
#endif lint
static char sccsid[] = "@(#)typein.c 1.5 87/01/07 Copyr 1986 Sun Micro";
#endif
*****/

#include <stdio.h>
#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <suntool/tty.h>
#include <ctype.h>

static Frame      frame;
static Canvas     canvas;
static Tty        tty;
static Pixwin     *pw;

static Notify_client my_client;

#define STDIN_FD      0
#define STDOUT_FD     1
#define BUFSIZE       1000

main(argc, argv)
int  argc;
char **argv;
{
    static Notify_value read_input();
    int  tty_fd;

    frame = window_create(NULL, FRAME,
        FRAME_ARGS,      argc, argv,
        WIN_ERROR_MSG,   "Cannot create frame",
        FRAME_LABEL,     "typein",
        0);

    tty = window_create(frame, TTY,
        WIN_PERCENT_HEIGHT, 50,
        TTY_ARGV,           TTY_ARGV_DO_NOT_FORK,
        0);

    tty_fd = (int>window_get(tty, TTY_TTY_FD);
    dup2(tty_fd, STDOUT_FD);
    dup2(tty_fd, STDIN_FD);

    canvas = window_create(frame, CANVAS,
        0);
    pw = canvas_pixwin(canvas);

    /*
     * Set up a notify proc so that whenever there is input to read on
     * stdin (fd 0), we are called to read it.
     * Notifier needs a unique handle: give it the address of tty.
     */
    my_client = (Notify_client) &tty;
    notify_set_input_func(my_client, read_input, STDIN_FD);

    printf("Enter first coordinate:\nx? ");
}

```

```

        window_main_loop(frame);
        exit(0);
    }

    /*
     * This section implements a simple application which writes prompts to
     * stdin and reads coordinates from stdout, drawing vectors with the
     * supplied coordinates. It uses a state machine to keep track of what
     * number to read next.
     */
#define GET_X_1      0
#define GET_Y_1      1
#define GET_X_2      2
#define GET_Y_2      3
int state = GET_X_1;
int x1, y1, x2, y2;

/* ARGSUSED */
static Notify_value
read_input(client, in_fd)
    Notify_client  client;          /* unused since this must be from ttysw */
    int            in_fd;           /* unused since this is stdin */
{
    char    buf[BUFSIZE];
    char    *ptr, *gets();

    ptr = gets(buf);    /* read one line per call so that we
                        don't ever block */
                        /* ^^^^^ does this matter any more?? */

    /* handle end of file */
    if (ptr==NULL) {
        /* Note: could have been a read error */
        window_set(frame, FRAME_NO_CONFIRM, TRUE, 0);
        window_done(tty);
    } else {
        switch (state) {
            case GET_X_1:
                if (sscanf(buf, "%d", &x1) != 1) {
                    printf("Illegal value!\nx? ");
                    fflush(stdout);
                } else {
                    printf("y? ");
                    fflush(stdout);
                    state++;
                }
                break;
            case GET_Y_1:
                if (sscanf(buf, "%d", &y1) != 1) {
                    printf("Illegal value!\ny? ");
                    fflush(stdout);
                } else {
                    printf("Enter second coordinate:\nx? ");
                    fflush(stdout);
                    state++;
                }
                break;
            case GET_X_2:

```



```
        if (sscanf(buf, "%d", &x2) != 1) {
            printf("Illegal value!\nx? ");
            fflush(stdout);
        } else {
            printf("y? ");
            fflush(stdout);
            state++;
        }
        break;
case GET_Y_2:
    if (sscanf(buf, "%d", &y2) != 1) {
        printf("Illegal value!\ny? ");
        fflush(stdout);
    } else {
        printf("Vector from (%d, %d) to (%d, %d)\n",
            x1, y1, x2, y2);
        pw_vector(pw, x1, y1, x2, y2, PIX_SET, 1);
        printf("\nEnter first coordinate:\nx? ");
        fflush(stdout);
        state = GET_X_1;
    }
    break;
}
return (NOTIFY_DONE);
}
```

A.9. Programs that Manipulate Color

The following two programs work with color. You can run them on a monochrome workstation to no ill-effect, but you won't see much of interest.

The techniques employed by these two programs are explained in the *Color* section of Chapter 7, *Imaging Facilities: Pixwins*.

When using these programs, try invoking them with different colors using the frame's command line arguments. Also, run *showcolor* (listed in the *pixwin* chapter) to see how the screen's colormap changes as different color programs are run simultaneously.

coloredit

The first program, *coloredit*, puts up sliders that let the user modify its colors.


```

/*****/
#ifndef lint
static char sccsid[] = "@(#)coloredit.c 1.4 86/09/15 Copyr 1986 Sun Micro";
#endif
/*****/

#include <suntool/sunview.h>
#include <suntool/panel.h>
#include <suntool/canvas.h>

#define MYFRAME      0
#define MYPANEL      1
#define MYCANVAS     2

/* colormap sizes for the three windows. Canvas is still the biggest */
mycms_sizes[3] = {
    2, 2, 4
};

#define MYCMS_SIZE    4
/* color arrays; initialize them with the canvas colors */
unsigned char  red[MYCMS_SIZE] = {0, 0, 255, 255};
unsigned char  green[MYCMS_SIZE] = {0, 255, 0, 192};
unsigned char  blue[MYCMS_SIZE] = {255, 0, 0, 192};

static void      getcms();
static void      setcms();
static void      cycle();
static void      editcms();
static void      set_color();
static void      change_value();

Panel_item      text_item;
Panel_item      color_item;
Panel_item      red_item, green_item, blue_item;

Pixwin          *pixwins[3];
Pixwin          *pw;

main(argc, argv)
    int          argc;
    char         **argv;
{
    Frame        base_frame;
    Panel        panel;
    Canvas       canvas;

    Attr_avlist  sliderdefaults;

    /* the cmsname is copied, so this array can be reused */
    char         cmsname[CMS_NAMESIZE];

    int          counter;
    int          xposition;
    char         buf[40];

    base_frame = window_create(NULL, FRAME,
                                FRAME_LABEL,
                                "coloredit",

```

```

                                FRAME_ARGS,      argc, argv,
                                0);

/* set up the panel */
panel = window_create(base_frame, PANEL,
                      0);
/* create a reusable attribute list for my slider attributes */
sliderdefaults = attr_create_list(
    PANEL_SHOW_ITEM,      TRUE,
    PANEL_MIN_VALUE,      0,
    PANEL_MAX_VALUE,      255,
    PANEL_SLIDER_WIDTH,   512,
    PANEL_SHOW_RANGE,     TRUE,
    PANEL_SHOW_VALUE,     TRUE,
    PANEL_NOTIFY_LEVEL,   PANEL_ALL,
    0);

panel_create_item(panel, PANEL_CYCLE,
    PANEL_LABEL_STRING,   "Edit colormap:",
    PANEL_VALUE,          MYCANVAS,
    PANEL_CHOICE_STRINGS, "Frame", "Panel", "Canvas", 0,
    PANEL_NOTIFY_PROC,    editcms,
    0);

text_item = panel_create_item(panel, PANEL_TEXT,
    PANEL_VALUE_DISPLAY_LENGTH, CMS_NAMESIZE,
    PANEL_VALUE_STORED_LENGTH,  CMS_NAMESIZE,
    0);

color_item = panel_create_item(panel, PANEL_SLIDER,
    ATTR_LIST,            sliderdefaults,
    PANEL_LABEL_STRING,    "color:",
    PANEL_NOTIFY_PROC,     set_color,
    0);

red_item = panel_create_item(panel, PANEL_SLIDER,
    ATTR_LIST,            sliderdefaults,
    PANEL_LABEL_STRING,    "  red:",
    PANEL_NOTIFY_PROC,     change_value,
    0);

green_item = panel_create_item(panel, PANEL_SLIDER,
    ATTR_LIST,            sliderdefaults,
    PANEL_LABEL_STRING,    "green:",
    PANEL_NOTIFY_PROC,     change_value,
    0);

blue_item = panel_create_item(panel, PANEL_SLIDER,
    ATTR_LIST,            sliderdefaults,
    PANEL_LABEL_STRING,    " blue:",
    PANEL_NOTIFY_PROC,     change_value,
    0);

panel_create_item(panel, PANEL_BUTTON,
    PANEL_LABEL_IMAGE,

```



```

        panel_button_image(panel, "Cycle colormap", 12, NULL),
        PANEL_NOTIFY_PROC, cycle,
        0);

window_fit(panel);
window_fit_width(base_frame);

/* free the slider attribute list */
free(sliderdefaults);

/* set up the canvas */
canvas = window_create(base_frame, CANVAS, 0);

/* get pixwins */
pixwins[MYFRAME] = (Pixwin *) window_get(base_frame, WIN_PIXWIN);
pixwins[MYPANEL] = (Pixwin *) window_get(panel, WIN_PIXWIN);
pw = pixwins[MYCANVAS] = (Pixwin *) canvas_pixwin(canvas);

/* set up the canvas' colormap */
sprintf(cmsname, "coloredit%D", getpid());
pw_setcmsname(pw, cmsname);
pw_putcolormap(pw, 0, mycms_sizes[MYCANVAS], red, green, blue);

/* draw in the canvas */
/* don't draw color 0 -- it is the background */
for (counter = 1; counter < mycms_sizes[MYCANVAS]; counter++) {
    xposition = counter * 100;
    pw_rop(pw, xposition, 50, 50, 50,
           PIX_SRC | PIX_COLOR(counter), NULL, 0, 0);
    sprintf(buf, "%d", counter);
    pw_text(pw, xposition + 5, 70, PIX_SRC ^ PIX_DST, 0, buf);
}
pw_text(pw, 100, 150,
        PIX_SRC | PIX_COLOR(mycms_sizes[MYCANVAS] - 1), 0,
        "This is written in the foreground color");

/* initialize to edit the first canvas color */
editcms(NULL, MYCANVAS, NULL);

window_main_loop(base_frame);
exit(0);
}

static int      cur_cms = -1;
/* ARGSUSED */
static void
editcms(item, value, event)
    Panel_item    item;
    unsigned int   value;
    Event          *event;
{
    int            planes;
    struct colormapseg cms;
    char           cmsname[CMS_NAMESIZE];

    if (value == cur_cms)
        return;

```

```

    cur_cms = value;
    /* get the new cmsname */
    pw_getcmsname(pixwins[cur_cms], cmsname);
    panel_set_value(text_item, cmsname);

    pw = pixwins[cur_cms];

    /* get the new colormap */
    /*
     * first have to get its size there is NO DOCUMENTED procedure to do
     * this.
     */
    pw_getcmsdata(pw, &cms, &planes);

    pw_getcolormap(pw, 0, cms.cms_size, red, green, blue);

    panel_set(color_item,
              PANEL_VALUE, 0,
              PANEL_MAX_VALUE, cms.cms_size - 1,
              0);
    /* call the proc to set the colors */
    set_color(NULL, 0, NULL);
}

int          cur_color;
/* ARGSUSED */
static void
set_color(item, color, event)
    Panel_item    item;
    unsigned int   color;
    struct inputevent *event;
{
    panel_set_value(red_item, red[color]);
    panel_set_value(green_item, green[color]);
    panel_set_value(blue_item, blue[color]);
    cur_color = (unsigned char) color;
}

/* ARGSUSED */
static void
change_value(item, value, event)
    Panel_item    item;
    int           value;
    struct inputevent *event;
{
    if (item == red_item)
        red[cur_color] = (unsigned char) value;
    else if (item == green_item)
        green[cur_color] = (unsigned char) value;
    else
        blue[cur_color] = (unsigned char) value;
    /*
     * pw_putcolormap expects arrays of colors, but this only sets one
     * color
     */
    pw_putcolormap(pw, cur_color, 1,
                   &red[cur_color], &green[cur_color], &blue[cur_color]);
}

```



```
}

/* ARGSUSED */
static void
cycle(item, event)
    Panel_item    item;
    Event         *event;
{
    pw_cyclecolormap(pw, 1, 0, mycms_sizes[cur_cms]);
}
```

animatecolor

This program demonstrates smooth animation via the technique of software double-buffering. Two colormaps for the canvas are set up so that while one is being written to, the other is being displayed. This allows smoother animation.

The routines that set up the colormaps and swap them, `doublebuff_init()` and `doublebuff_swap()`, are general enough to be used in other programs that alternate two colormaps. You need only set up a similar `colorstuff` structure to use these routines in another program.

The logic involved in creating the colormaps is complex. The colormaps created for *animatecolor* are given in the table *Sample Colormap to Isolate Planes* in the *pixwin* chapter.


```

/*****
#endif lint
static char sccsid[] = "@(#)animatecolor.c 1.4 88/03/09 Copyr 1986 Sun Micro";
#endif
*****/

#include <suntool/sunview.h>
#include <suntool/canvas.h>

/*****
/* You set MYCOLORS & MYNBITS according to how many colors      */
/* you are using; rest is just boilerplate, more or less;        */
/* it you define your colors.                                     */
*****/
/*
 * define the colors I want in the canvas; max 16, must be a
 * power of 2
 */
#define MYCOLORS          4
/*
 * define the number of bits my colors take up -- MYCOLORS log 2;
 * maximum for animation to be possible is half screen's bits per
 * pixel -- 4 bits on current Sun color displays.
 */
#define MYNBITS            2
/*
 * to "hide" one set of planes while displaying another takes a
 * large cms -- the square of the number of colors
 */
#define MYCMS_SIZE        (MYCOLORS * MYCOLORS)

/*
 * when you write out a color pixel, you must write the color in
 * the appropriate planes. This macro writes it in both sets
 */
#define usecolor(i)        ( (i) | ((i) << colorstuff.colorbits) )

struct colorstuff {
    /* desired colors */
    unsigned char    redcolors[MYCOLORS];
    unsigned char    greencolors[MYCOLORS];
    unsigned char    bluecolors[MYCOLORS];
    /* number of bits the desired colors take up */
    int              colorbits;
    /* colormap segment size */
    int              cms_size;
    /* 2 colormaps to support it */
    unsigned char    red[2][MYCMS_SIZE];
    unsigned char    green[2][MYCMS_SIZE];
    unsigned char    blue[2][MYCMS_SIZE];
    /* 2 masks to support it */
    int              enable_0_mask;
    int              enable_1_mask;
    /* current colormap -- 0 or 1 */
    int              cur_buff;
    /* plane mask to control which planes are written to */
    int              plane_mask;

```

```

};

struct colorstuff colorstuff = {
/* desired red colors */
                                {0, 0, 255, 255},
/* desired green colors */
                                {0, 255, 0, 192},
/* desired blue colors */
                                {255, 0, 0, 192},
/* number of planes these colors take up */
                                MYNBITS,
/* colormap segment size */
                                MYCMS_SIZE,
/* rest filled in later */
};

static void    resize_proc();

/* stuff needed to do random numbers */
extern void    srand();
extern int     getpid();
extern long    random();
extern char    *sprintf();

static Notify_value my_frame_interposer();
static Notify_value my_draw();

static Pixwin  *pw;
static int      times_drawn;
static int      Xmax, Ymax;

main(argc, argv)
    int      argc;
    char      **argv;
{
    Frame      base_frame;
    Canvas     canvas;

    base_frame = window_create(NULL, FRAME,
                                FRAME_LABEL, "animatecolor",
                                FRAME_ARGS, argc, argv,
                                0);
    canvas = window_create(base_frame, CANVAS,
                            CANVAS_RETAINED, TRUE,
                            CANVAS_RESIZE_PROC, resize_proc,
                            0);
    pw = (Pixwin *) canvas_pixwin(canvas);

    /* set up the canvas' colormap */
    doublebuff_init(&colorstuff);

    /* run the drawing routine as often as possible */
    (void) notify_set_itimer_func(base_frame, my_draw,
                                ITIMER_REAL,
                                &NOTIFY_POLLING_ITIMER,
                                ((struct itimerval *) 0));

    /* initialize the random function */

```



```

    srand(getpid());
    window_main_loop(base_frame);
    exit(0);
}

/* ARGSUSED */
static      Notify_value
my_draw(client, itimer_type)
    Notify_client  client;
    int            itimer_type;
{
    /*
     * draw the squares, then swap the colormap to animate them
     */
#define SQUARESIZE      50
#define MAX_VEL          (SQUARESIZE / 5)
    /* number of squares to animate */
#define NUMBER          (MYCOLORS - 1)

    static int          posx[NUMBER], posy[NUMBER];
    static int          vx[NUMBER], vy[NUMBER];
    static int          prevposx[NUMBER], prevposy[NUMBER];
    int                 i;

    /* set the plane mask to be that which we are not viewing */
    pw_putattributes(pw, (colorstuff.cur_buff == 1) ?
        &(colorstuff.enable_1_mask) : &(colorstuff.enable_0_mask));

    /* write to invisible planes */
    for (i = 0; i < NUMBER; i++) {
        if (!times_drawn) {
            /* first time drawing */

            posx[i] = (i + 1) * 100;
            posy[i] = 50;
            vx[i] = r(-MAX_VEL, MAX_VEL);
            vy[i] = r(-MAX_VEL, MAX_VEL);
        }
        if (abs(vx[i]) > MAX_VEL) {
            printf("Weird value (%d) for vx[%d]\n", vx[i], i);
            vx[i] = r(-MAX_VEL, MAX_VEL);
        }
        posx[i] = posx[i] + vx[i];
        if (posx[i] < 0) {
            /* Bounce off the left wall */
            posx[i] = 0;
            vx[i] = -vx[i];
        } else if (posx[i] > Xmax - SQUARESIZE) {
            /* Bounce off the right wall */
            vx[i] = -vx[i];
            posx[i] = posx[i] + vx[i];
        }
        posy[i] = posy[i] + vy[i];
        if (posy[i] > Ymax - SQUARESIZE) {
            /* Bounce off the top */
            posy[i] = Ymax - SQUARESIZE;
            vy[i] = -vy[i];
        } else if (posy[i] < 0) {

```

```

        /* Bounce off the bottom */
        posy[i] = 0;
        vy[i] = -vy[i];
    }
    /* draw the square you can't see */
    pw_rop(pw, posx[i], posy[i], SQUARESIZE, SQUARESIZE,
           PIX_SRC | PIX_COLOR(usecolor(i + 1)), NULL, 0, 0);
}
/*
 * swap the colormaps, and hey presto! should appear smoothly
 */
doublebuff_swap(&colorstuff);
times_drawn++;

/* set the plane mask to be that which we are not viewing */
pw_putattributes(pw, (colorstuff.cur_buff == 1) ?
                 &(colorstuff.enable_1_mask): &(colorstuff.enable_0_mask));

/* erase now invisible planes */
for (i = 0; i < NUMBER; i++) {

    if (times_drawn > 1) {
        /* squares have been drawn before */
        /* erase in the one you can't see */
        pw_rop(pw, prevposx[i], prevposy[i],
               SQUARESIZE, SQUARESIZE, PIX_CLR, NULL, 0, 0);
    }
    /* remember so can erase later */
    prevposx[i] = posx[i];
    prevposy[i] = posy[i];
}

/*
 * set the plane mask to be that which we are viewing, in
 * case screen has to be repaired between now an when we are
 * called again.
 */
pw_putattributes(pw, (colorstuff.cur_buff == 1) ?
                 &(colorstuff.enable_0_mask): &(colorstuff.enable_1_mask));
}

/* random number calculator */
int
r(minr, maxr)
    int          minr, maxr;
{
    int          i;

    i = random() % (maxr - minr + 1);
    if (i < 0)
        return (i + maxr + 1);
    else
        return (i + minr);
}

```



```

/* ARGSUNUSED */
static void
resize_proc(canvas, width, height)
{
    times_drawn = 0;
    /* remember, pixels start at 0, not 1, in the pixwin */
    Xmax = width - 1;
    Ymax = height - 1;
}

/*
 * Do double buffering by changing the write enable planes and
 * the color maps. The application draws into a buffer which is
 * not visible and when the buffers are swapped the invisible one
 * become visible and the other become invis.
 *
 * Start out drawing into buffer 1 which is the low-order buffer;
 * ie. the low-order planes. Things would not work if this is not
 * done because the devices start out be drawing with color 1
 * which will only hit the low-order planes.
 *
 * Init double buffering: Allocate color maps for both buffers. Fill
 * in color maps.
 */

doublebuff_init(colorstuff)
    struct colorstuff *colorstuff;
{
    /*
     * user has defined desired colors. Set them up in the two
     * colormap segments
     */
    int          index_1;
    int          index_2;
    int          i;
    char         cmsname[CMS_NAMESIZE];

    /* name colormap something unique */
    sprintf(cmsname, "animatecolor%D", getpid());
    pw_setcmsname(pw, cmsname);

    /*
     * for each index in each color table, figure out how it maps
     * into the original color table.
     */
    for (i = 0; i < colorstuff->cms_size; i++) {
        /*
         * first colormap will show color X whenever low order
         * bits of color index are X
         */
        index_1 = i & ((1 << colorstuff->colorbits) - 1);
        /*
         * second colormap will show color X whenever high order
         * bits of color index are X
         */
        index_2 = i >> colorstuff->colorbits;
    }
}

```

```

    colorstuff->red[0][i] = colorstuff->redcolors[index_1];
    colorstuff->green[0][i] = colorstuff->greencolors[index_1];
    colorstuff->blue[0][i] = colorstuff->bluecolors[index_1];

    colorstuff->red[1][i] = colorstuff->redcolors[index_2];
    colorstuff->green[1][i] = colorstuff->greencolors[index_2];
    colorstuff->blue[1][i] = colorstuff->bluecolors[index_2];
}
colorstuff->enable_1_mask = ((1 << colorstuff->colorbits) - 1)
    << colorstuff->colorbits;
colorstuff->enable_0_mask = ((1 << colorstuff->colorbits) - 1);

/*
 * doublebuff_swap sets up the colormap. We want the drawing
 * to start off drawing into the 1st buffer, so set the
 * current buffer to 1 so that when doublebuff_swap is called
 * it will set up the first ([0] ) colormap.
 */
colorstuff->cur_buff == 1;
doublebuff_swap(colorstuff);
}

/*
 * Routine to swap buffers by loading a color map that will show
 * the contents of the buffer that was not visible. Also, set the
 * write enable plane so that future writes will only effect the
 * planes which are not visible.
 */
doublebuff_swap(colorstuff)
    struct colorstuff *colorstuff;
{
    if (colorstuff->cur_buff == 0) {
        /* display first buffer while writing to 2nd */
        /*
         * Careful! pw_putcolormap() wants an array or pointer
         * passed, but the colormap arrays are 2-d
         */
        pw_putcolormap(pw, 0, colorstuff->cms_size,
            colorstuff->red[0],
            colorstuff->green[0],
            colorstuff->blue[0]);
        /* set plane mask to write to second buffer */
        colorstuff->plane_mask = colorstuff->enable_1_mask;
        colorstuff->cur_buff = 1;
    } else {
        /* display second buffer while writing to first */
        pw_putcolormap(pw, 0, colorstuff->cms_size,
            colorstuff->red[1],
            colorstuff->green[1],
            colorstuff->blue[1]);

        /* set plane mask to write to first buffer */
        colorstuff->plane_mask = colorstuff->enable_0_mask;
        colorstuff->cur_buff = 0;
    }
}

```


**A.10. Two gfx
subwindow-based
programs converted
to use SunView**

The following two programs are the Sun demo programs *bouncedemo* and *spheresdemo* converted from using `gfxsw_init()` to canvases in SunView.

The code for the SunWindows-based programs is in `/usr/share/src/sun/suntool` so you can contrast that code with the SunView versions printed here.

Techniques used to convert programs such as these to SunView 1 are described in Appendix C, *Converting SunWindows Programs to SunView*.

bounce

The first program is *bouncedemo* converted to draw in a canvas and to call `notify_dispatch()` periodically. Like the original *bouncedemo*, it restarts drawing after any damage (if not retained) or resizing.

```

#ifndef lint
static char sccsid[] = "@(#)bounce.c 1.5 88/02/26 Copyr 1986 Sun Micro";
#endif

/*
 * Overview:    Bouncing ball demo in window.
 * Converted to use SunView by simulating the gfxsubwindow structure.
 */

/* this replaces all includes */
#include <suntool/sunview.h>
#include <suntool/canvas.h>

/* straight from the Canvas chapter */
static void      repaint_proc();
static void      resize_proc();

/* straight from the Notifier chapter */
static Notify_value my_notice_destroy();
extern Notify_error notify_dispatch();

static int      my_done;

/* define my own gfxsubwindow struct */
struct gfxsubwindow {
    int          gfx_flags;
#define GFX_RESTART    0x01
#define GFX_DAMAGED    0x02
    int          gfx_reps;
    struct pixwin *gfx_pixwin;
    struct rect   gfx_rect;
}
    mygfx;
struct gfxsubwindow *gfx = &mygfx;

```



```

main(argc, argv)
    int      argc;
    char     **argv;
{
    short     x, y, vx, vy, z, ylastcount, ylast;
    short     Xmax, Ymax, size;
    /* WIN_RECT attribute returns a pointer */
    Rect      *rect;

    /* have to handle this arg that gfxsw_init used to process */
    int        retained;

    /*
     * replace this call if (gfx == (struct gfxsubwindow *)0) exit(1);
     * with ...
     */

    Frame      frame;
    Canvas     canvas;
    Pixwin     *pw;

    /* this arg was also dealt with by gfxsw_init */
    gfx->gfx_reps = 200000;

    frame = window_create(NULL, FRAME,
                          FRAME_LABEL, "bounce",
                          FRAME_ARGC_PTR_ARGV, &argc, argv,
                          WIN_ERROR_MSG, "Can't create frame",
                          0);
    for (--argc, ++argv; *argv; argv++) {
        /*
         * handle the arguments that gfxsw_init(0, argv) used to do
         * for you
         */
        if (strcmp(*argv, "-r") == 0)
            retained = 1;
        if (strcmp(*argv, "-n") == 0)
            if (argc > 1) {
                (void) sscanf(*(++argv), "%hD", &gfx->gfx_reps);
                argc++;
            }
    }

    canvas = window_create(frame, CANVAS,
                          CANVAS_RETAINED, retained,
                          CANVAS_RESIZE_PROC, resize_proc,
                          CANVAS_FAST_MONO, TRUE,
                          WIN_ERROR_MSG, "Can't create canvas",
                          0);

    /* only need to define a repaint proc if not retained */
    if (!retained) {
        window_set(canvas,
                  CANVAS_REPAINT_PROC, repaint_proc,
                  0);
    }
    pw = canvas_pixwin(canvas);
}

```

```
gfx->gfx_pixwin = canvas_pixwin(canvas);

/* Interpose my proc so I know that the tool is going away. */
(void) notify_interpose_destroy_func(frame, my_notice_destroy);

/*
 * Note: instead of window_main_loop, just show the frame. The
 * drawing loop is in control, not the notifier.
 */
window_set(frame, WIN_SHOW, TRUE, 0);
```


Restart:

```

rect = (Rect *) window_get(canvas, WIN_RECT);
Xmax = rect_right(rect);
Ymax = rect_bottom(rect);
if (Xmax < Ymax)
    size = Xmax / 29 + 1;
else
    size = Ymax / 29 + 1;
/*
 * the following were always 0 in a gfx subwindow (bouncedemo
 * is confused on this point
 */
x = 0;
y = 0;

vx = 4;
vy = 0;
ylast = 0;
ylastcount = 0;
pw_writebackground(pw, 0, 0, rect->r_width, rect->r_height,
    PIX_SRC);
/*
 * Call notify_dispatch() to dispatch events to the frame
 * regularly. This will call my_resize and repaint procs and
 * interposed notify_destroy_func if necessary. The latter will
 * set my_done to TRUE if it's time to finish.
 */
while (gfx->gfx_reps) {
    (void) notify_dispatch();
    if (my_done)
        break;
    /*
     * this program is not concerned with damage, because either
     * the canvas repairs the damage (if retained) or it just
     * restarts, which is handled by GFX_RESTART
     */
    /*
     * if (gfx->gfx_flags & GFX_DAMAGED) gfxsw_handlesigwinch(gfx);
     */
    if (gfx->gfx_flags & GFX_RESTART) {
        gfx->gfx_flags &= ~GFX_RESTART;
        goto Restart;
    }
    if (y == ylast) {
        if (ylastcount++ > 5)
            goto Reset;
    } else {
        ylast = y;
        ylastcount = 0;
    }
    pw_writebackground(pw, x, y, size, size,
        PIX_NOT(PIX_DST));
    x = x + vx;
    if (x > (Xmax - size)) {
        /*
         * Bounce off the right edge
         */
        x = 2 * (Xmax - size) - x;

```

```

        vx = -vx;
    } else if (x < 0) {
        /*
         * bounce off the left edge
         */
        x = -x;
        vx = -vx;
    }
    vy = vy + 1;
    y = y + vy;
    if (y >= (Ymax - size)) {
        /*
         * bounce off the bottom edge
         */
        y = Ymax - size;
        if (vy < size)
            vy = 1 - vy;
        else
            vy = vy / size - vy;
        if (vy == 0)
            goto Reset;
    }
    for (z = 0; z <= 1000; z++);
    continue;

Reset:
    if (--gfx->gfx_reps <= 0)
        break;
    x = 0;
    y = 0;
    vx = 4;
    vy = 0;
    ylast = 0;
    ylastcount = 0;
}
}

```



```
static void
repaint_proc( /* Ignore args */ )
{
    /* if repainting is required, just restart */
    gfx->gfx_flags |= GFX_RESTART;
}

static void
resize_proc( /* Ignore args */ )
{
    gfx->gfx_flags |= GFX_RESTART;
}

/* this is straight from the Notifier chapter */
static Notify_value
my_notice_destroy(frame, status)
    Frame      frame;
    Destroy_status status;
{
    if (status != DESTROY_CHECKING) {
        /* set my flag so that I terminate my loop soon */
        my_done = 1;
        /* Stop the notifier if blocked on read or select */
        (void) notify_stop();
    }
    /* Let frame get destroy event */
    return (notify_next_destroy_func(frame, status));
}
```

spheres

This is an example of a program that has been converted to use `window_main_loop()`. It displays a fixed-sized image in a canvas that has scrollbars. It continues drawing its image when its window is damaged or resized. However, it stops drawing when it is iconic.

You will have to create your own icon for this called `spheres.icon`.


```

#ifndef lint
static char sccsid[] = "@(#)spheres.c 1.4 88/02/05 Copyr 1986 Sun Micro";
#endif
/*
 * spheres -- draw a bunch of shaded spheres Algorithm was done
 * by Tom Duff, Lucasfilm Ltd., 1982
 * Revised to use SunView canvas instead of gfxsw.
 */

#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <suntool/scrollbar.h>
#include <sunwindow/cms_rainbow.h>

static Notify_value my_frame_interposer();
static Notify_value my_animation();
static void sphere();
static void demoflushbuf();

#define ITIMER_NULL ((struct itimerval *)0)

/*
 * (NX, NY, NZ) is the light source vector -- length should be
 * 100
 */
#define NX 48
#define NY -36
#define NZ 80

#define BUF_BITWIDTH 16

static struct pixrect *mpr;
static int width;
static int height;
static int counter;
static Frame frame;
static Canvas canvas;
static int cmssize;
static Pixwin *pw;

static short spheres_image[256] = {
#include "spheres.icon"
};

mpr_static(spheres_pixrect, 64, 64, 1, spheres_image);

main(argc, argv)
    int argc;
    char **argv;
{
    char **args;
    int usefullgray = 0;
    Icon icon;

    icon = icon_create(ICON_IMAGE, &spheres_pixrect, 0);
    frame = window_create(NULL, FRAME,
                          FRAME_LABEL, "spheres",

```

```

        FRAME_ICON,          icon,
        FRAME_ARGC_PTR_ARGV, &argc, argv,
        0);
canvas = window_create(frame, CANVAS,
    CANVAS_AUTO_EXPAND,    0,
    CANVAS_AUTO_SHRINK,    0,
    CANVAS_AUTO_CLEAR,     0,
    /*
     * Set SCROLL_LINE_HEIGHT to 1 so that clicking LEFT or RIGHT
     * in the scroll buttons scrolls the canvas by one pixel.
     */
    WIN_VERTICAL_SCROLLBAR, scrollbar_create(SCROLL_LINE_HEIGHT, 1,
                                              0),
    WIN_HORIZONTAL_SCROLLBAR, scrollbar_create(SCROLL_LINE_HEIGHT, 1,
                                              0),
    0);
for (args = argv; *args; args++) {
    if (strcmp(*args, "-g") == 0)
        usefullgray = 1;
}
/* Interpose in front of the frame's client event handler */
(void) notify_interpose_event_func(frame, my_frame_interposer,
    NOTIFY_SAFE);
(void) notify_set_itimer_func(frame, my_animation,
    ITIMER_REAL, &NOTIFY_POLLING_ITIMER, ITIMER_NULL);

width = (int) window_get(canvas, CANVAS_WIDTH);
height = (int) window_get(canvas, CANVAS_HEIGHT);
pw = canvas_pixwin(canvas);
cmssize = (usefullgray) ? setupfullgraycolormap(pw) :
    setuprainbowcolormap(pw);
mpr = mem_create(BUF_BITWIDTH, height, pw->pw_pixrect->pr_depth);
window_main_loop(frame);
exit(0);
}

static int    radius;
static int    x0;          /* x center */
static int    y0;          /* y center */
static int    color;
static int    x;
static int    y;
static int    maxy;
static int    mark;
static int    xbuf;

/* ARGSUSED */
static      Notify_value
my_animation(client, itimer_type)
    Notify_client    client;
    int              itimer_type;
{
    register        i;

    if (x >= radius) {
        radius = r(0, min(width / 2, height / 2));
        x0 = r(0, width);
        y0 = r(0, height);
    }
}

```



```

    color = r(0, cmssize + counter++) % cmssize;
    x = -radius;
    xbuf = 0;

    /*
     * Don't use background colored sphere.
     */
    if (color == 0)
        color++;
    /*
     * Don't use tiny sphere.
     */
    if (radius < 8)
        radius = 8;
}
for (i = 0; i < 5; i++) {
    xbuf++;
    maxy = sqroot(radius * radius - x * x);
    pw_vector(pw, x0 + x, y0 - maxy, x0 + x, y0 + maxy,
              PIX_CLR, 0);
    for (y = -maxy; y <= maxy; y++) {
        mark = r(0, radius * 100) <= NX * x + NY * y
              + NZ * sqroot(radius * radius - x * x - y * y);
        if (mark)
            pr_put(mpr, xbuf, y + y0, color);
    }
    if (xbuf == (mpr->pr_width - 1)) {
        demoflushbuf(mpr, PIX_SRC | PIX_DST,
                     x + x0 - mpr->pr_width, pw);
        xbuf = 0;
        x++;
        return (NOTIFY_DONE);
    }
    x++;
}
if (x >= radius)
    demoflushbuf(mpr, PIX_SRC | PIX_DST, x + x0 - (xbuf + 2),
                 pw);
return (NOTIFY_DONE);
}

static void
demoflushbuf(mpr, op, x, pixwin)
    struct pixrect *mpr;
    int             op;
    int             x;
    struct pixwin   *pixwin;
{
    register u_char *sptr, *end;

    sptr = mprd8_addr(mpr_d(mpr), 0, 0, mpr->pr_depth);
    end = mprd8_addr(mpr_d(mpr), mpr->pr_width - 1,
                     mpr->pr_height - 1, mpr->pr_depth);
    /*
     * Flush the mpr to the pixwin.
     */
    pw_write(pixwin, x, 0, mpr->pr_width, mpr->pr_height, op,

```

```

        mpr, 0, 0);

/*
 * Clear mpr with 0's
 */
while (sptr <= end)
    *sptr++ = 0;
/* Let user interact with tool */
notify_dispatch();
}

static int
setuprainbowcolormap(pw)
    Pixwin      *pw;
{
    register u_char red[CMS_RAINBOWSIZE];
    register u_char green[CMS_RAINBOWSIZE];
    register u_char blue[CMS_RAINBOWSIZE];

    /*
     * Initialize to rainbow cms.
     */
    pw_setcmsname(pw, CMS_RAINBOW);
    cms_rainbowsetup(red, green, blue);
    pw_putcolormap(pw, 0, CMS_RAINBOWSIZE, red, green, blue);
    return (CMS_RAINBOWSIZE);
}

static int
setupfullgraycolormap(pw)
    Pixwin      *pw;
{
#define CMS_FULLGRAYSIZE      256
#define CMS_FULLGRAY          "fullgray"
    register u_char red[CMS_FULLGRAYSIZE];
    register u_char green[CMS_FULLGRAYSIZE];
    register u_char blue[CMS_FULLGRAYSIZE];
    register      i;

    /*
     * Initialize to rainbow cms.
     */
    pw_setcmsname(pw, CMS_FULLGRAY);
    for (i = 0; i < CMS_FULLGRAYSIZE; i++) {
        red[i] = green[i] = blue[i] = i;
    }
    pw_putcolormap(pw, 0, CMS_FULLGRAYSIZE, red, green, blue);
    return (CMS_FULLGRAYSIZE);
}

static      Notify_value
my_frame_interposer(frame, event, arg, type)
    Frame      frame;
    Event      *event;
    Notify_arg  arg;
    Notify_event_type type;
{

```



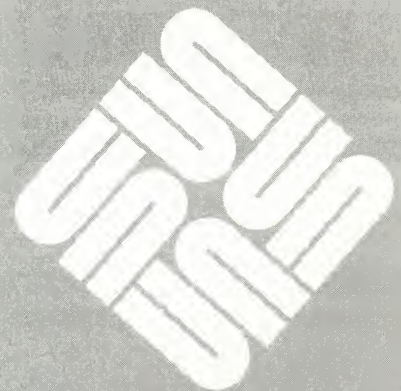
```
int      closed_initial, closed_current;
Notify_value  value;

/* Determine initial state of frame */
closed_initial = (int) window_get(frame, FRAME_CLOSED);
/* Let frame operate on the event */
value = notify_next_event_func(frame, event, arg, type);
/* Determine current state of frame */
closed_current = (int) window_get(frame, FRAME_CLOSED);
/* Change animation if states differ */
if (closed_initial != closed_current) {
    if (closed_current) {
        /* Turn off animation because closed */
        (void) notify_set_itimer_func(frame, my_animation,
                                      ITIMER_REAL, ITIMER_NULL, ITIMER_NULL);
    } else {
        /* Turn on animation because opened */
        (void) notify_set_itimer_func(frame, my_animation,
                                      ITIMER_REAL, &NOTIFY_POLLING_ITIMER,
                                      ITIMER_NULL);
    }
}
return (value);
}
```

B

Sun User Interface Conventions

Sun User Interface Conventions	469
B.1. Program Names	469
B.2. Frame Headers	469
B.3. Menus	469
Capitalization	469
Menus Showing Button Modifiers	470
Interaction with Standard Menus	470
Enable/Disable Menu Items	470
Multi-Column Menus	470
B.4. Panels	470
Buttons	471
List of Non-Exclusive Choices	471
List of Exclusive Choices	471
Binary Choices	472
Text Items	472
Allocation of Function Between Buttons and Menus	472
B.5. Mouse Button Usage	473
Allocation of Function Between Mouse Buttons	473
Using Mouse Buttons for Accelerators	473
B.6. Cursors	473
B.7. Icons	473



B

Sun User Interface Conventions

The window programs released by Sun follow some standard user interface conventions. These conventions are described here so that, if you choose, you can design your interfaces with them in mind.

B.1. Program Names

Here are some guidelines for naming programs:

- A window-based version of an existing tty-based program has *tool* appended to the end of the existing program. For example `mailtool` is a window-based version of the tty-based program `mail(1)`.
- A program without a tty version should not end with *tool*. Thus the icon editor is called `iconedit` and not *icontool*.
- Since tools are normally invoked from command files or menus, descriptive names are better than short cryptic ones. Thus *iconedit* is better than *ied*.

B.2. Frame Headers

The frame header should contain the name of the program, optionally followed by a dash and additional information, as in:

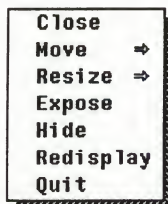


```
textedit - /tmp/file, dir: /usr/dg/doc
```

B.3. Menus

Capitalization

The words in menus should be capitalized as they would be in a chapter heading:



```
Close
Move  =>
Resize =>
Expose
Hide
Redisplay
Quit
```

This convention can be bent when the names in the menu correspond to already existing, non-capitalized command names.

Menus Showing Button Modifiers

When the behavior of a panel button depends on whether the user holds down a shift key, the button should have a menu summarizing the different actions, as in this menu from the **Reply** button in **mailtool**:

```
reply
Reply (all)                [Shift]
reply, include             [Ctrl]
Reply (all), include       [Ctrl][Shift]
```

Interaction with Standard Menus

Standard SunView menus, such as the frame menu, should not be modified. When a user is used to seeing 'Quit' at the end of the frame menu, it is confusing to see a frame menu with a new item tacked on at the end. Equally confusing is a frame menu that comes up with an item other than 'Close' at the top. Thus, instead of deleting an item from a standard menu, applications should render the item inactive and "grayed-out." And instead of adding a new item to a standard menu, applications should make a new menu, with the name of the standard menu at the top, followed by the application-specific commands. The standard menu then becomes a pullright subordinate to the custom menu, as in the example below:

Frame	Close
Dump Scre	Move →
Dump Regl	Resize →
Print Dum	Expose
View Dump	Hide
	Redisplay
	Quit

Enable/Disable Menu Items

Sometimes a menu has two different states, with different words appearing in the same position in a menu, depending on the current state. When the two states correspond to something being on or off, the words 'Enable' and 'Disable' should be used. Thus **shelltool** uses 'Enable Page Mode' and 'Disable Page Mode'.

Multi-Column Menus

Overly long menus should be avoided. Use menus with more than one column instead.

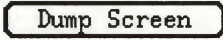
B.4. Panels

The defaults for panel items given in this section are intended to promote consistency across applications and provide convenient building blocks for programmers who don't want to put a great deal of effort into designing fancy panels. The intent is not to rule out the use of non-default panel items.

Buttons

The proper use of buttons is to allow the user to initiate commands. Button items should not be used to represent categories, modes or options — for these kinds of choices that imply a change of state, you should use toggle, choice or cycle items, as described in the next three sections.

When creating a button, use the routine `panel_button_image()` to create a button-like image, as in:

 Dump Screen

As with menu entries, capitalize buttons unless the button name matches something else (for example, `dbx(1)` commands in `dbxtool`). If the button's meaning can be modified by **Control** or **Shift** these modifiers should be indicated in the button's menu. (For an example, see the picture of the **Reply** menu from `mailtool`, at the top of the preceding page.)

In most cases, a button will remain visible all the time. However, when a tool has different states, and a button can only be used in some of those states, it is usually best to make the button invisible when it can not be invoked. Thus in `mailtool`, the **Cancel** button only appears when a letter is being composed.

List of Non-Exclusive Choices

A list of choices in which more than one choice can be selected at a time is best implemented with the item type `PANEL_TOGGLE`. The default for toggles is a list of check boxes:

Optional Software:

- ☒ Database
- ☐ Demos
- ☒ Document Preparation Tools
- ☐ Games
- ☒ Productivity Tools

The example shows a vertical list; vertical or horizontal are both acceptable.

List of Exclusive Choices

A list of choices in which only one choice can be selected at a time can be displayed with all choices visible or with only the current choice visible. To show all the choices, use the item type `PANEL_CHOICE`. The default for choice items is a list of square pushbuttons, with the current choice marked by a darkened pushbutton:

Drawing Mode: ☐ Point ☒ Line ☐ Rectangle ☐ Circle ☐ Text

To show only the current choice, use `PANEL_CYCLE`. This item type provides a symbol consisting of two circular arrows, which indicate to the user that he can cycle through choices, and serves to distinguish cycle items from text items:

Category  SunView

Binary Choices

An item that is either on or off may be created using either `PANEL_TOGGLE`, `PANEL_CYCLE` or `PANEL_CHOICE`. The picture on the left below is a toggle, the two in the middle are cycles, and the one on the right is a choice:

☒ Grid Show Grid  Yes Grid  On Grid:  On  Off

Text Items

Text items should have a colon after the label.

For text items, it is recommended to have one or more buttons which cause the text item's value to be acted on. In `iconedit`, for example, the user first enters a filename into the **File:** field, then presses the **Load**, **Store**, or **Browse** button in order to act on that filename.

`iconedit` also allows the user to type `(Control-L)`, `(Control-S)`, or `(Control-B)` into the **File:** field as accelerators for the buttons. Use of such accelerators (including carriage return to mean "enter") is *not* recommended, as it conflicts with future plans for the use of non-printable characters.

For the sake of consistency, whenever a tool reads from and writes to a file, it should label these buttons with **Load** and **Store**.

Allocation of Function Between Buttons and Menus

Selecting a menu item is normally the same as either selecting a button or picking from a choice item. `boggetool(6)`, for example, has a menu for restarting the game (as well as other things) but has no buttons. Each of the four menu items could have been represented by a button instead. `life(6)` does not have a choice item, but rather lets you choose a starting pattern with a menu. Thus the question of when to use a button (or choice) and when to use a menu arises. Here are some rules of thumb:

- Items on the frame menu should not be duplicated as buttons, with the possible exception of a **Quit** button (see next paragraph).
- Some tools typically run all the time, such as `mailtool`. Others are normally invoked only long enough to do a job, such as `iconedit`. Tools in the second category, if they have any other buttons, should also have a **Quit** button.
- If a tool has a commit operation, then it may have a **Done** button, which is a combination of close¹⁰⁶ plus commit. Thus `mailtool` has a **Done** button.
- A tool should never have a **Close** button, since this operation is already available via both a menu and the keyboard.
- If a custom menu is provided, the menu items should not all be duplicated as panel items (buttons or choices). `boggetool` and `life` are examples of programs that have functionality in custom menus that are not duplicated as panel items.
- When a button and a menu item perform the same function, their labels should be identical.

¹⁰⁶ If the panel is in a subframe, the **Done** operation implies disappearing from the screen rather than closing, since subframes can't become iconic.

B.5. Mouse Button Usage

Allocation of Function Between Mouse Buttons

Use of mouse buttons should be consistent with the rest of SunView. The left button should only be used to make selections. The right button should only be used to bring up menus.¹⁰⁷

There is some discretion involved in the use of the middle button, however. In most of SunView, the middle button is used to adjust a selection. In text and shell windows, for example, the left button is used to mark the starting point of a selection, and the middle button is used to extend the selection. Similarly, in a pixel editor that allowed you to select regions, clicking the left button on a region could select just that region, and clicking the middle button on another region could add that region to the selection. On the other hand, in a tool that allowed you to move objects, the middle button could move an object, and **[Control]-MIDDLE** button could re-size it, which would be consistent with the way icons and frames are moved and re-sized. As a third alternative, in the *iconedit* drawing program the left button draws pixels (which is a kind of selecting) and the middle button erases.

The best use of the middle button is still being discussed. Future versions of this guideline may specify more exactly how the middle button should be used. For now, the most common use is to extend the selection, and the next-most common is to move a graphic object.

Using Mouse Buttons for Accelerators

It is acceptable to use the mouse buttons as accelerators for common operations. The only caveat is that any accelerators should also be available from a menu or panel item. Thus in SunView clicking on a tool with the middle button moves the tool, but you can also move a tool using the frame menu.

Some operations, on the other hand, cannot be invoked from a menu or panel button. In such cases the mouse is the only means of invoking the operation. For example, in *iconedit* you use the mouse for drawing, and the drawing operations are not available from a menu or button.

B.6. Cursors

An application program should not do anything other than change the shape of the cursor when the cursor is moved into a new window. *textedit* presents a good example of using the cursor to alert the user that input is interpreted differently in different regions: The cursor is a thin diagonal arrow in the *textsubwindow*, a fat horizontal arrow in the scrollbar, and a diamond in the scrollbar buttons.

B.7. Icons

Tools should pack as much information as possible into their icons. *clock* and *perfmer* are examples of tools that make good use of icon real estate. *textedit* is an example of a tool that could make better use of its icon. For example, it could contain a representation of the text being editing in a 1 point font. Small as that is, you can tell at a glance if you are editing C code or a mail message.

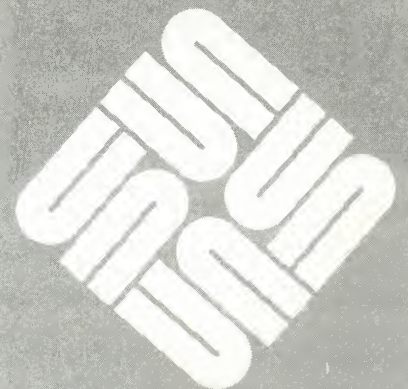
¹⁰⁷ People who want to hold the mouse with their left hand can put the "menu button" on the left and the "select button" on the right by setting the *Left_Handed* option in the *Input* category of *defaultsedit*.

Some icons, like the round face used by `clock` and the page with the protruding pencil used by `textedit`, have images with non-square outlines. These icons have the area outside of the image outline filled in with the root grey pattern so that the icons will blend in with the default SunView background. While this looks good when the background is in fact the default pattern, it is not recommended, since users can choose an arbitrary background pattern for SunView.

C

Converting SunWindows Programs to SunView

Converting SunWindows Programs to SunView	477
C.1. Converting Tools	478
General Comments	478
Programming Style Changes	478
Object typedefs	478
Attribute Value Interface	478
New Objects	479
Canvas Subwindows	479
Text Subwindows	479
Scrollbars	479
Objects in Common between SunView and SunWindows	480
Cursors	480
Icons	480
Menus	481
Input Events	481
Setting up Input Event Handling	482
Sigwinch Handling	482
Windows	482
Panels	482
Signals	483
Prompts	483
C.2. Converting Gfxsubwindow-Based Code	485
Basic Steps	485



Replacing Tool Interaction	485
Styles of Damage Checking	485
<i>Either</i> the Notifier Takes Over	485
<i>Or</i> Your Code Stays in Control	486
Handling Damage	486
The gfxsw Structure	486
Finishing Up	487
Miscellaneous	487
Two Examples	488

Converting SunWindows Programs to SunView

This appendix gives some guidelines for converting programs written using SunWindows to SunView. There are two classes of programs covered: those that create a tool and subwindows, and programs that call `gfxsw_init()` to take over an existing window or the console.

Programs that fall outside these classes are probably UNIX-style programs that do not use windows at all. The conversion of such programs is in effect the subject of this whole manual. If you want to convert such a program to SunView, pay particular attention to Chapter 2, *The SunView Model*, and the specific discussion of Notifier interaction under *Porting Programs to SunView* in Chapter 17, *The Notifier*. You may also find some of the discussion later on in this appendix under Section C.2, *Converting Gfxsubwindow-Based Code*, helpful.

C.1. Converting Tools

It is reasonably straightforward to convert tools that create windows in SunWindows to the SunView interface because they should already have the appropriate architecture. SunView programs, like SunWindows programs, have three parts: initialization of static objects, starting up window system interaction, and the routines that are called after the tool is running in the window system.

General Comments

When porting to SunView, you should look through all of your code for SunWindows function calls. If you see one, the odds are that you are going to have others. Look for every occurrence of the call and then change it to the new format. Since the SunView libraries are mixed in with the SunWindows libraries, you can mix the two types of functions calls, and not get any compilation errors. But you *will* get some inconsistent results.

Programming Style Changes

Object typedefs

The capitalized typedefs for window system objects (applied to Panels, Panel_items and Panel_settings in 2.0 SunWindows) have been extended to nearly all SunView objects, including:

Canvas	Pixrect
Cursor	Pixwin
Frame	Rect
Icon	Rectlist
Menu	Scrollbar
Panel	Textsw
Panel_item	Tty
Pixfont	Window

You should convert to using these data types in the interests of future compatibility. See *Object Handles* in Chapter 3, *Interface Outline*, for more information on these types.

Attribute Value Interface

In SunView, the attribute value interface, introduced for panel subwindows in 2.0 SunWindows, has been extended to all types of windows. Attributes for all window types are set and obtained with the same two calls, `window_set()` and `window_get()`.

All window types are created with the same call, `window_create()`.

CAUTION

The most frequently used SunView calls use attribute lists, and therefore must be null-terminated. SunView will only complain about a malformed attribute list at run time.

New Objects

Most of the data types in the above list are objects new in SunView. Many objects in SunWindows correspond to objects in SunView, for example:

```
tool    ⇒    Frame
ttysw   ⇒    Tty
```

Some objects such as the graphics subwindow and empty subwindow are not supported in SunView¹⁰⁸. There are new objects that partially take their place.

Canvas Subwindows

The canvas subwindow is a general-purpose drawing subwindow, which can replace gfx subwindows and empty subwindows. The size of the canvas you draw on need not be the same as the size of the window it is displayed in; you can create scrollbars to let the user adjust the visible part of the canvas. For a demonstration of the various canvas attributes, run the program */usr/demo/canvas_demo*

Text Subwindows

These allow for the display and editing of text in a scrollable window. The user can perform various actions on the text, including saving the text, searching in the text, and editing the text without the programmer having to deal with these interactions.

Since there was no such window in SunWindows, your application may have had to use a gfx subwindow, a set of panel message items, or some strange technique involving `ttysw_input()` or piping to a tty subwindow to display text; the text subwindow can replace all these uses.

Scrollbars

Scrollbars can be attached to windows. In particular, the use of scrollbars with retained canvases makes it very easy to draw a fixed-size image without regard for window size changes.

¹⁰⁸ You can still compile and run code that uses these, but Sun does not intend to develop them further.

Objects in Common between SunView and SunWindows

Cursors

Cursors have changed. They are now type `Cursor`, and all calls relating to them have changed. Type `Cursor` should be looked at as a pointer to the structure containing the cursor information. Here is how you would define a cursor:

```
static short int help_bits [] = {
#include "help.curs"
};
mpr_static(help_pr, 16, 16, 1, help_bits);
```

Once having created a cursor, you call `window_set()` to add it to a window, as in the following code fragment:

```
Cursor help_cursor;

main()
{
    /* make windows */
    ...
    init_cursor();
    ...
}

init_cursor()
{
    help_cursor = cursor_create(CURSOR_IMAGE, &help_pr,
                                CURSOR_XHOT, 8,
                                CURSOR_YHOT, 8,
                                CURSOR_OP, XOR,
                                0);
    window_set(window, WIN_CURSOR, help_cursor, 0);
}
```

You now refer to all your cursors by the handle you get from `cursor_create()`. Cursors have their own create, destroy, copy, set, and get routines, as well as a number of attributes with no corresponding functionality in SunWindows.

Icons

Icons have changed. They follow the same pattern as cursors; you define the data, create a `pixrect`, and then call `icon_create()` at run time. These also have their own create, destroy, set and get routines, although there are fewer attributes associated with them.

Menus

The new walking menu package uses the attribute value interface. It has many more features than the old menu package. It does not support the stacking menu style of SunWindows.¹⁰⁹

Menus also have their own routines and are created via function calls instead of being user-loaded data structures. They use the pointer type `Menu` for their handles instead of `struct menuptr`. One way to create them is to write a special `menu_init()` proc which loads them into their structures correctly. In your `menu_init()`, you have something like

```
ml_items = menu_create(
    MENU_STRING_ITEM, "insert",    INSERT,
    MENU_STRING_ITEM, "copy",      COPY ,
    MENU_STRING_ITEM, "replace",    REPLACE,
    MENU_STRING_ITEM, "move",       XLATE ,
    MENU_STRING_ITEM, "delete",     DELETE,
    MENU_STRING_ITEM, "HELP",       DRAW_HELP,
    0);
```

Menu values from `menu_get()` or `menu_show()` are returned as `caddr_t`'s. Be sure your types match.

NOTE The old `menu_display()` and the new `menu_show()` routines have a different order for the arguments.

Input Events

The `inputevent` structure has not changed. However, you no longer have to generate events yourself in "selected routines via calls to `input_readevent()`. Instead, windows now have event handlers that are passed pointers to Event structures.

There are a number of macros for making input events easier to deal with in SunView, so instead of having something like `ie->ie_code` you have `event_id(ie)`, resulting in more readable code.

Event types are not pointers, so you have to distinguish between

```
Event *ie;
```

and

```
Event ie;
```

in your code. You can use either, because the event functions don't just manipulate a handle as, for example, the cursor functions do. See *Object Handles* in Chapter 3, *Interface Outline*, for an explanation of when handles are pointers and when not.

¹⁰⁹ This is still available in the frame and root menus if you disable *SunView/Walking_Menus* in `defaultsedit`.

Setting up Input Event Handling

All the input events can be set up from the `window_create()` call or `window_set()` calls. Calls to `win_*inputmask()` are all replaced by these `window_set()` and `window_create()` calls.

The distinction between "pick" and "keyboard" events is new in SunView, having been added to support the notion of a split input focus.

CAUTION

Be careful that when you are setting mouse events, you are modifying the `WIN_*_PICK_EVENTS` and when you are setting keyboard events you modify `WIN_*_KEYBOARD_EVENTS`. You may get inconsistent results if you modify pick events on the keyboard mask.

Sigwinch Handling

Canvas event procedures no longer need all the gfx support for flag checking. Resize and repaint events are separately handled by the procedures you supply via the `CANVAS_RESIZE_PROC` and `CANVAS_REPAINT_PROC` attributes. These procedures mean you should not try to catch sigwinch signals (and in fact, if you do, you will have problems; see below).

Windows

Making windows is very straightforward in SunView. Each window type has a handle — so instead of the inconsistent use of handles and fd's to describe a window and manipulate it, you only use the window handle. You need to go through your code and update all the reference to the old `tool_...` handle types in the code. After you find them, locate all the function calls referring to them and update them to SunView `window_set()` and `window_get()` calls. Almost every window operation is supported by the attribute value interface; however, some low-level routines that are documented in the *SunView 1 System Programmer's Guide* may still require window names or fd's.

`window_get()` is used to get an attribute of a window. It returns a `caddr_t` back to you, which must be cast into the appropriate type. So loading something into a `rect` struct would involve something like:

```
Rect    win_size;
Canvas  canvas;

canvas = window_create( base_frame, CANVAS, 0);
win_size= *((Rect *)window_get(canvas, WIN_RECT));
```

NOTE

Be sure to cast values returned from `get()` routines to the correct type.

The above `*((Rect *)...)` is needed otherwise you will get an 'incompatible type' message from the compiler.

Panels

Most of the panel interface was already using an attribute value interface in 2.0 SunWindows. `panel_create()` `panel_set()` and `panel_get()` should be changed to `window_create()`, `window_set()` and `window_get()`.

The `PANEL_CU()` macro was superseded by `ATTR_COL()` and `ATTR_ROW()`.

Signals

If you are catching signals, then you should read the documentation on signals in Section 17.2, *Restrictions*, in the *Notifier* chapter. There are several that the Notifier now catches on your behalf.

You should no longer be catching SIGWINCH signals. If you do, your program may never appear on the screen as it will start catching the signals and redrawing endlessly on the screen, which may not be visible.

Prompts

Instead of using the `menu_prompt()` facility of SunWindows, you should use the alerts package to prompt the user, or if necessary use pop-up subframes and `window_loop(popup_frame)` when prompting the user. The *filer* example programs in Chapter 4, *Windows*, uses the alerts package to implement a pop-up confirmer.

`menu_prompt()` is documented here for completeness. The definitions used by `menu_prompt()` are:

```
struct prompt {
    Rect    prt_rect;
    Pixfont *prt_font;
    char    *prt_text;
}
```

```
menu_prompt(prompt, event, windowfd)
    struct prompt    *prompt;
    struct inputevent *event;
    int              windowfd;
```

`menu_prompt()` displays the string addressed by `prompt->prt_text` using the font `prompt->prt_font`. `prompt->prt_rect` is relative to `windowfd`. If either the `r_width` or the `r_height` fields of `prompt->prt_rect` has the value `PROMPT_FLEXIBLE`, that dimension is chosen to accommodate all the characters in `prompt->prt_text`.

The fullscreen access method is used to display the prompt. After displaying the prompt, `menu_prompt()` waits for any input event other than mouse motion. It then removes the prompt, and returns the event which caused the return in event. `windowfd` is the file descriptor of the window from which input is taken while the prompt is up.

Table C-1 *SunWindows \Rightarrow SunView Equivalences*

<i>In SunWindows</i>	<i>In Sunview</i>
<code>tool = tool_make()</code>	<code>Frame frame = window_create(NULL, FRAME, ..., 0);</code>
<code>tool_parse_all</code>	<code>FRAME_ARGS</code> or <code>FRAME_ARGC_PTR_ARGV</code> attributes to <code>window_create(NULL, FRAME, ..., 0)</code>
<code>tool_install()</code> <code>tool_select()</code> <code>tool_destroy()</code>	<code>window_main_loop(frame);</code>
or, individually,	
<code>tool_install()</code>	<code>WIN_SHOW</code> attribute
<code>tool_select()</code>	<code>window_main_loop()</code> , <code>notify_dispatch()</code> or <code>notify_start()</code>
<code>tool_destroy()</code>	<code>window_destroy(baseframe)</code> or <code>window_done(any_window)</code>
<code>signal(SIGWINCH, sigwinch)</code>	<code>RESIZE_PROC</code> and <code>REPAINT_PROC</code> attribute
<code>TOOLSW_EXTENDTOEDGE</code>	<code>WIN_EXTEND_TO_EDGE</code>
<code>win_grabio()</code>	<code>WIN_GRAB_ALL_INPUT</code> attribute
<code>struct tool_io</code>	<code>WIN_EVENT_PROC</code> for window events. Other events, timers, etc. handled by individual calls to the Notifier to set up or interpose specific procs.

C.2. Converting Gfxsubwindow-Based Code

Programs that run in gfxsubwindows are designed to take over an existing window. In SunView you must create a tool for such programs to run in. One limitation of this approach is that the SunView version of the application must run under `suntools`; the old `gfxsw_init()` call would create a SunWindows environment if run on the “bare” Sun console. One major advantage gained by moving to SunView is that your code can use scrollbars.

Basic Steps

- Include `<suntool/sunview.h>` and `<suntool/canvas.h>`.
- Remove all window-related `#include` statements; these will probably be included by `sunview.h`.
- Declare a `Frame` and a `Canvas`.
- Replace `gfxsw_init()` with calls to create the frame and canvas.

Replacing Tool Interaction

Styles of Damage Checking

Many gfx subwindow programs (and many of the Sun demos) call `gfxsw_init()` to take over a window, then run in a loop as they compute and draw an image in the gfx subwindow. At some point in the loop they check for damage to or alteration of the size of the gfx subwindow and handle it accordingly.

In SunView, the coexistence of your program with the window system is less hidden from you. Read Chapter 2, *The SunView Model*, to understand how this coexistence works. In converting programs, you must ensure the Notifier runs at regular intervals so that window events such as close, quit, etc. are handled appropriately.

Consult Chapter 17, *The Notifier*, for more information.

You can either (1) set up your program so that, after initialization, control passes to the Notifier, which you have set up to call your imaging/computation routine periodically, or (2) let control continue to pass to your code, and change the program to call the Notifier at regular intervals.

Either the Notifier Takes Over

In the first case, you set up your imaging/computation routine as a function that is called when a timer expires. Do this by calling `notify_set_itimer_func()`. If you want your imaging/computation routine to blaze away non-stop (causing other programs to run more sluggishly), you request the timer function be called as soon as the Notifier has handled window events for you by giving the timer the special value `&NOTIFY_POLLING_ITIMER`.

```
(void) notify_set_itimer_func(frame, my_animation,
                             ITIMER_REAL, &NOTIFY_POLLING_ITIMER, ITIMER_NULL);
```

If your code `sleep()` 's on a regular basis, then you should be able to modify it so that the Notifier calls your imaging/computation routine at the same interval.

The program *spheres* in Appendix A, *Example Programs*, is an example of this style of interaction.

Or Your Code Stays in Control

On the other hand, if your program just loops, perhaps while (`--gfx_reps`), then you could add to the loop a call to `notify_dispatch()`. This will handle window system events and return.

The program *bounce* in Appendix A, *Example Programs*, is an example of this style of interaction.

If you do this then your program has to detect when the user has 'Quit' from the menu: see *Finishing Up* below.

NOTE `gfx_reps` in a `gfx` subwindow program is set to a large number (200,000), but the user can change it through the command line argument `-n number_of_repetitions`.

Handling Damage

The Notifier will handle moving the window, resizing it, etc. However, resulting damage to your canvas may need to be repaired. In the `gfx` subwindow, `GFX_DAMAGED` is set whenever a `SIGWINCH` is received. In addition `GFX_RESTART` is set if the size of the window has changed or if the window is not retained. `GFX_DAMAGED` is set as a hint for you to call `gfxsw_handlesigwinch()`, which would clear up the damaged list and if the window was retained it would repaint the image for you. `GFX_RESTART` is set as a hint that the window had to be rebuilt, either because of damage and the window is not retained, or because of a resize.

Many situations that you would need to handle yourself in a `gfx` subwindow are rendered superfluous by attributes of the canvas subwindow, such as `CANVAS_AUTO_CLEAR`, etc. For starters, canvases are retained by default; if your canvas has scrollbars and is retained, then you need not be concerned with resize events. Nevertheless, you may need to be aware when you must rebuild or repair your image. Read the *Canvases* chapter for more information.

Rather than setting a flag, SunView calls your own procedure if you specify one with the `CANVAS_REPAINT_PROC` and `CANVAS_RESIZE_PROC` attributes. These are called with useful parameters for their tasks.

You can modify your code so that the repair activity that used to take place after noticing the flags have been set now takes place in the procs themselves, or you can write the procs so they set flags similar to the `GFX_RESTART` and `GFX_REPAINT` flags and return, and leave your repair code almost untouched. Or, depending on your application, you can set up your canvas so that the window system handles all damage.

The `gfxsw` Structure

The `gfxsw` structure has fields in it that carry useful information. Comparable information is available in SunView, so you can declare and setup a comparable structure in SunView. The *bounce* program in Appendix A, *Example Programs*, does this.

`Gfx` subwindow-based programs use the `gfx->gfxsw_rect` to determine the geometry of the window they are drawing in. Since the starting point of this was

relative to the `gfxsw`, it was always 0.¹¹⁰ In SunView the width and height of the canvas you draw in are available through the canvas attributes `CANVAS_WIDTH` and `CANVAS_HEIGHT`. The fields of the `gfx->gfxsw_rect` correspond to these attributes as follows:

<code>coord</code>	<code>r_left, r_top;</code>	are both = 0
<code>short</code>	<code>r_width, r_height;</code>	are the <code>CANVAS_WIDTH</code> and <code>CANVAS_HEIGHT</code> attributes.

As described above, you can use your own `GFX_RESTART` and `GFX_REPAINT` flags.

If you care about the `gfxsw` command line arguments, insert code into your program's `argv, argc` parsing loop to handle the `gfx` options that used to be taken care of for you. The *bounce* program has reasonable code to do this.

Finishing Up

If your imaging routine is in control and periodically calls the Notifier, then when the window is quit your routine must know that this has occurred. Otherwise, the imaging routine will continue to draw in a window that has been destroyed, and you will see error messages like

```
WIN ioctl number C0146720: Bad file number
```

until you kill the program.

What you must do is interpose in front of the frame's destroy event handler so that your program will know when the frame goes away. See the item on *Getting Out* in *Porting Programs to SunView* in the *Notifier* chapter.

If your program exits on its own, then it can call `window_done()` to destroy its windows. This will invoke your interposed notice-destroy routine (which may or may not matter depending on what it does). It will also call the standard

```
Are you sure you want to Quit?
```

alert unless you set `FRAME_NO_CONFIRM`.

Miscellaneous

`gfxsw_getretained()` is equivalent to the `CANVAS_RETAINED` attribute. Canvases are retained by default.

`gfxsw_init()` doesn't consume the `gfxsw` command line options `-r`, `-n` *Number_of_repetitions*, etc; your code may do strange things with its arguments to deal with this.

¹¹⁰ Many of the demos supplied by Sun are confused on this point and go through unnecessary steps.

Two Examples

Listings of two programs converted from SunView are in Appendix A, *Example Programs*.

bounce

The first is a new version of *bouncedemo*(6). It now draws its bouncing square in a canvas. It has code to parse the standard *gfx* subwindow command line arguments. It preserves the original `while (gfx->gfx_reps) {...}` loop structure of *bouncedemo* by calling `notify_dispatch()` at the top of the loop. Because it is running in a loop it must detect when the user has 'Quit' the window, so it interposes before its frame's destroy routine using `notify_interpose_destroy_func()`. The routine that is called just sets a flag so the program knows to exit from the loop.

spheres

The second is a version of *spheresdemo*(6). It now draws its shaded spheres in a canvas with scrollbars, so you can see all the image in a small window. It handles the notification of SunView events by asking the Notifier to call the drawing routine (`my_animation()`) as often as possible, using

```
(void) notify_set_itimer_func(frame, my_animation,
                             ITIMER_REAL, &NOTIFY_POLLING_ITIMER, ITIMER_NULL);
```

Since the drawing operation is under the control of the Notifier, the Notifier can control the program, so the `while (gfx->gfx_reps) {...}` loop structure is replaced by a call to `window_main_loop()`; this will terminate the program when the user chooses 'Quit' from the frame menu.

Detecting when the Program is Iconic

spheres detects when it is made iconic by interposing in front of the frame's client event handler using `notify_interpose_event_func()`. The routine that is called calls the normal `event_func`, then checks to see if the frame has changed state: if it has been closed it turns the notify timer off altogether, so the drawing routine is no longer called; if it has been opened the timer is set back to immediate polling.

bounce should do this also — there is little point in drawing when iconic unless you are drawing a single compute-intensive image.

Index

A

- action procedure for menu item, 229
- alarm(3), 285
- alert, 199
 - arrow, 201
 - beeping, 202
 - buttons, 201
 - components of, 201
 - controlling beeping, 208
 - creation, 202
 - described, 199
 - example program, 389
 - interface summary, 316
 - messages and simple buttons, 203
 - position of, 202
 - table of attributes, 316
 - table of functions, 318
 - text message, 201
 - types of buttons, 204
 - use with blocking pop-ups, 47
 - uses of, 201
 - using trigger events, 206
- alert attributes, 316
 - ALERT_BUTTON, 316
 - ALERT_BUTTON_FONT, 316
 - ALERT_BUTTON_NO, 204, 316
 - ALERT_BUTTON_YES, 204, 316
 - ALERT_MESSAGE, 203
 - ALERT_MESSAGE_FONT, 316
 - ALERT_MESSAGE_STRINGS, 316
 - ALERT_MESSAGE_STRINGS_ARRAY_PTR, 316
 - ALERT_NO_BEEPING, 208, 316
 - ALERT_OPTIONAL, 317
 - ALERT_POSITION, 317
 - ALERT_TRIGGER, 207, 317
- alert functions, 318
 - alert_prompt(), 202, 318
 - possible status values, 202
- ASCII events, 86
- asynchronous signal notification, 293
- ATTR_COL, 54, 160, 309
- ATTR_COLS, 310
- ATTR_LIST, 311
- ATTR_ROW, 54, 160, 309
- ATTR_ROWS, 310
- attribute functions
 - attr_create_list(), 310

attribute lists

- creating reusable lists, 310
- default attributes, 311
- maximum size, 29, 311
- overview, 28
- utilities, 309

attribute ordering, 55

- for canvases, 69
- in text subwindow, 132

B

- base frame, 16
- boundary manager, 19
- button image constructor, 168
- button panel item, 158, 167 *thru* 169
- buttons with menus, 168

C

- callback procs, 20
- callback style of programming, 20
- canvas, 61
 - attribute order, 69
 - automatic sizing, 69
 - backing pixrect, 66
 - canvas space vs. window space, 70, 98
 - color in canvases, 72
 - coordinate system, 65
 - default input mask, 70
 - definition of, 61
 - handling input, 70
 - interface summary, 319
 - model, 65
 - monochrome in multiple plane groups, 125
 - non-retained, 66
 - pixwin, 63, 65
 - repaint procedure, 66
 - repainting, 66
 - resize procedure, 67
 - retained, 66
 - scrolling, 64
 - table of attributes, 319
 - table of functions and macros, 320
 - tracking changes in size, 67
 - writing your own event procedure, 70
- canvas attributes, 319
 - CANVAS_AUTO_CLEAR, 66, 319
 - CANVAS_AUTO_EXPAND, 69, 319
 - CANVAS_AUTO_SHRINK, 69, 319

- canvas attributes, *continued*
 - CANVAS_FAST_MONO, 319
 - CANVAS_FIXED_IMAGE, 67, 319
 - CANVAS_HEIGHT, 65, 319
 - CANVAS_MARGIN, 65, 319
 - CANVAS_PIXWIN, 65, 103, 319
 - CANVAS_REPAINT_PROC, 66, 319
 - CANVAS_RESIZE_PROC, 319
 - CANVAS_RETAINED, 66, 319
 - CANVAS_WIDTH, 65, 69, 319
- canvas functions and macros, 320
 - canvas_event(), 71, 98, 320
 - canvas_pixwin(), 63, 65, 103, 320
 - canvas_window_event(), 70, 320
- CAPSMASK, 96, 97
- character unit macros
 - ATTR_COL, 54, 160, 309
 - ATTR_COLS, 310
 - ATTR_ROW, 54, 160, 309
 - ATTR_ROWS, 310
- child process control using the Notifier, 288
- choice panel item, 158, 170 *thru* 176
- classes of windows, 16
- client handle used by the Notifier, 287
- clipping in a pixwin, 112
- code examples, *see* example programs
- color, 113
 - advanced colormap manipulation example program, 441
 - animation, 122, 447
 - background color of pixwin, 115
 - color during fullscreen access, 119
 - colormap, 113
 - colormap access, 118
 - colormap segment, 114
 - cursors and menus, 119
 - default colormap segment, 115
 - determining if display is color, 119
 - example programs, 441
 - fast color change, 114
 - foreground color of pixwin, 115
 - FRAME_BACKGROUND_COLOR, 115
 - FRAME_FOREGROUND_COLOR, 115
 - FRAME_INHERIT_COLORS, 115
 - grayscale compatibility, 120
 - hardware double-buffering, 122
 - in canvases, 72
 - introduction, 113
 - one colormap segment per window, 116
 - showcolor, 116
 - software double-buffering, 120, 122
 - software double-buffering example program, 447
 - table of color functions, 360
 - using color, 119
- compiling SunView programs, 27
- confirmation
 - FRAME_NO_CONFIRM, 382
- control structure in Notifier-based programs, 21
- converting existing programs to use the Notifier, 303
- converting programs to SunView
 - attribute-value interface, 478
 - cursors, 480
 - equivalent SunWindows routines, 484
 - converting programs to SunView, *continued*
 - gfx subwindow-based, 485
 - gfx subwindow-based examples, 454
 - icons, 480
 - input events, 481
 - menus, 481
 - new objects, 479
 - non-window based programs, 477
 - panels, 482
 - prompts, 483
 - signals, 483
 - SunWindows-based, 477
 - typedefs, 478
 - windows, 482
 - write a prompt... read a reply, 216
- CTRLMASK, 96, 97
- cursor, 253
 - crosshair border gravity, 258
 - crosshair gap, 258
 - crosshair length, 258
 - crosshairs, 256
 - definition of, 253
 - fullscreen crosshairs, 258
 - hot spot, 257
 - interface summary, 321
 - rasterop, 257
 - setting position of mouse cursor, 91
 - table of attributes, 321
 - table of functions, 323
- cursor attributes, 321
 - CURSOR_CROSSHAIR_BORDER_GRAVITY, 258, 321
 - CURSOR_CROSSHAIR_COLOR, 321
 - CURSOR_CROSSHAIR_GAP, 258, 321
 - CURSOR_CROSSHAIR_LENGTH, 258
 - CURSOR_CROSSHAIR_OP, 321
 - CURSOR_CROSSHAIR_THICKNESS, 321
 - CURSOR_FULLSCREEN, 258, 321
 - CURSOR_HORIZ_HAIR_BORDER_GRAVITY, 321
 - CURSOR_HORIZ_HAIR_COLOR, 321
 - CURSOR_HORIZ_HAIR_GAP, 321
 - CURSOR_HORIZ_HAIR_OP, 321
 - CURSOR_HORIZ_HAIR_THICKNESS, 321
 - CURSOR_IMAGE, 256, 257, 321
 - CURSOR_OP, 257, 321
 - CURSOR_SHOW_CROSSHAIRS, 321
 - CURSOR_SHOW_CURSOR, 321
 - CURSOR_SHOW_HORIZ_HAIR, 321
 - CURSOR_SHOW_VERT_HAIR, 321
 - CURSOR_VERT_HAIR_BORDER_GRAVITY, 322
 - CURSOR_VERT_HAIR_COLOR, 322
 - CURSOR_VERT_HAIR_GAP, 322
 - CURSOR_VERT_HAIR_OP, 322
 - CURSOR_VERT_HAIR_THICKNESS, 322
 - CURSOR_XHOT, 257, 322
 - CURSOR_YHOT, 257, 322
- cursor constants
 - CURSOR_TO_EDGE, 258
- cursor functions, 323
 - cursor_copy(), 255, 323
 - cursor_create(), 255, 323
 - cursor_destroy(), 255, 323
 - cursor_get(), 255, 323
 - cursor_set(), 255, 323

CURSOR_TO_EDGE, 258
cycle panel item, 174

D

data types, 324
 caddr_t, 28
 object handles, 28
 opaque, 28
default colormap segment, 115
default system font, 41
DEFINE_ICON_FROM_IMAGE(), 40, 262
DESTROY_CHECKING, 300
DESTROY_CLEANUP, 300
DESTROY_PROCESS_DEATH, 300
destroying windows
 FRAME_NO_CONFIRM, 382
destruction of objects, 300
disable Quit confirmation
 FRAME_NO_CONFIRM, 41, 382
dispatching events
 calling the Notifier explicitly, 303
 calling the Notifier implicitly, 303
display
 batching, 110
 canvases and gfxsw's in multiple plane groups, 125
 determining if in color, 119
 enable plane, 124
 locking, 109
 locking and batching interaction, 112
 overlay plane, 124
 plane group, 124
 software double-buffering, 120
 speed, 108
distribution of input in a window, 90

E

event
 ASCII event codes, 86
 canvas space vs. window space, 70, 98
 definition of, 80
 function key event codes, 88
 keyboard focus event codes, 88
 META event codes, 86
 mouse button event codes, 86
 mouse motion event codes, 86
 panel space vs. window space, 193
 reading events explicitly, 97
 relationship to Notifier, 20
 repaint and resize event codes, 87
 shift key event codes, 89
 timeout, 294
 using an event with alerts, 206
 window entry and window exit event codes, 87
event codes, 330, 82
 BUT(), 86
 KBD_DONE, 88
 KBD_REQUEST, 88
 KBD_USE, 88
 KEY_LEFT, 88
 KEY_RIGHT, 88
 KEY_TOP, 88
 LOC_DRAG, 70, 86

event codes, *continued*

LOC_MOVE, 86
LOC_RGNENTER, 87
LOC_RGNEXIT, 87
LOC_STILL, 86
LOC_TRAJECTORY, 86
LOC_WINENTER, 87
LOC_WINEXIT, 87
MS_LEFT, 86
MS_MIDDLE, 86
MS_RIGHT, 86
PANEL_EVENT_CANCEL, 190
PANEL_EVENT_DRAG_IN, 190
PANEL_EVENT_MOVE_IN, 190
SCROLL_REQUEST, 80
SHIFT_CAPSLOCK, 89
SHIFT_CTRL, 89
SHIFT_LEFT, 89
SHIFT_LOCK, 89
SHIFT_META, 89
SHIFT_RIGHT, 89
WIN_REPAINT, 70, 87
WIN_RESIZE, 70, 87, 299
WIN_STOP, 88

event descriptors, 333

WIN_ASCII_EVENTS, 70, 90, 333
WIN_IN_TRANSIT_EVENTS, 90, 333
WIN_MOUSE_BUTTONS, 90, 333
WIN_NO_EVENTS, 90, 333
WIN_UP_ASCII_EVENTS, 90, 333
WIN_UP_EVENTS, 90, 333

event handling

at the Notifier level, 288
in canvases, 70
in panels, 189

event procedure

form of, 81
writing your own for a canvas, 70
writing your own for a panel item, 191

event state retrieval macros

event_action(), 96
event_ctrl_is_down(), 96
event_is_ascii(), 96
event_is_button(), 96
event_is_down(), 96
event_is_key_left(), 96
event_is_key_right(), 96
event_is_key_top(), 96
event_is_meta(), 96
event_is_up(), 96
event_meta_is_down(), 96
event_shift_is_down(), 96
event_shiftmask(), 96
event_time(), 96
event_x(), 96
event_y(), 96

event state setting macros

event_set_down(), 97
event_set_id(), 97
event_set_shiftmask(), 97
event_set_time(), 97
event_set_up(), 97
event_set_x(), 97

event state setting macros, *continued*

event_set_y(), 97

event stream, 80

example programs, 389

animatecolor, 122, 447

bounce, 454

color manipulation, 441

coloredit, 118, 441

colormap manipulation, 441

converting terminal-based programs, 216

creating menus, 416

dctool, 430

discussion of image_browser_1 program, 50

discussion of image_browser_2, 53

discussion of simple file manager, 44

filer, 44, 389

font_menu, 416

gfx subwindow-based demos converted to SunView, 454

gfxsw_init to SunView, 454, 461

image_browser_1, 401

image_browser_2, 406

list files in tty subwindow, 42

minimal SunView program, 37

notify_dispatch(), 216, 454

notify_set_itimer_func(), 461

resize_demo, 52, 299, 425

row/column space in a window, 406

showcolor, 116

simple file manager, 389

simple panel window, 39

spheres, 461

subwindow layout, 401

tty subwindow escape sequences, 412

tty subwindow I/O, 412

tty_io, 412

typein, 437

F

fcntl(2), 286, 290

file descriptor usage, 57

filer, 44

flow of control in Notifier-based programs, 21

font functions

pf_default(), 41, 106

pf_open(), 41

frame

command line frame attributes, 386

definition of, 16

fitting around subwindows, 41

frame header, 18

layout of subwindows within a frame, 51, 52, 299, 425

menus, 18

modifying destruction using the Notifier, 299

modifying open/close using the Notifier, 297

table of attributes, 382

frame attributes, 382

FRAME_ARGC_PTR_ARGV, 40, 382

FRAME_ARGS, 55, 382

FRAME_BACKGROUND_COLOR, 115, 382

FRAME_CLOSED, 382

FRAME_CLOSED_RECT, 382

FRAME_CMDLINE_HELP_PROC, 382

FRAME_CURRENT_RECT, 382

frame attributes, *continued*

FRAME_DEFAULT_DONE_PROC, 382

FRAME_DONE_PROC, 382

FRAME_EMBOLDEN_LABEL, 382

FRAME_FOREGROUND_COLOR, 115, 382

FRAME_ICON, 40, 382

FRAME_INHERIT_COLORS, 115, 382

FRAME_LABEL, 40, 382

FRAME_NO_CONFIRM, 41, 382

FRAME_NTH_SUBFRAME, 383

FRAME_NTH_SUBWINDOW, 383

FRAME_NTH_WINDOW, 383

FRAME_OPEN_RECT, 383

FRAME_SHOW_LABEL, 40, 45, 383

FRAME_SUBWINDOWS_ADJUSTABLE, 383

free(3), 311

function keys, 88

G

generate procedure

for menu, 244

for menu item, 243

for pull-right, 246

generate procedure operation parameter values

MENU_DISPLAY, 242

MENU_DISPLAY_DONE, 242

MENU_NOTIFY, 242

MENU_NOTIFY_DONE, 242

getitimer(2), 285

gfx subwindow

pw_use_fast_monochrome(), 125

converted demo programs, 454, 461

converting to SunView, 485

demo programs converted to SunView, 454

monochrome in multiple plane groups, 125

H

header files

overview, 27

<suntool/canvas.h>, 61

<suntool/icon.h>, 261

<suntool/menu.h>, 221

<suntool/panel.h>, 153

<suntool/scrollbar.h>, 165, 267

<suntool/seln.h>, 279

<suntool/sunview.h>, 27

<suntool/textsw.h>, 129

<suntool/tty.h>, 211

<sunwindow/attr.h>, 311

<sunwindow/cms_mono.h>, 115

<sunwindow/pixwin.h>, 101

<sunwindow/rect.h>, 52

<sunwindow/win_cursor.h>, 253

<sunwindow/win_input.h>, 77

<sunwindow/window_hs.h>, 77, 101

I

icon, 261

definition of, 19

interactive editor for icon images, 262

interface summary, 328

Loading Icon Images At Run Time, 263

modifying the icon's image, 263

icon, continued

- table of attributes, 328
- table of functions and macros, 329

icon attributes, 328

- ICON_FONT, 328
- ICON_HEIGHT, 328
- ICON_IMAGE, 328
- ICON_IMAGE_RECT, 328
- ICON_LABEL, 328
- ICON_LABEL_RECT, 328
- ICON_WIDTH, 328

icon functions and macros, 329

- DEFINE_ICON_FROM_IMAGE(), 40, 262, 329
- icon_create(), 262, 329
- icon_destroy(), 329
- icon_get(), 329
- icon_set(), 329

*image_browser_1, 50**image_browser_2, 53**images*

- in icons, 262
- in menus, 227
- mpr_static(), 227, 262
- using images generated with iconedit, 227, 262

*include files — see “header files”, 27**initiating event processing, 35**input, 77*

- ASCII events, 90, 333
- designee, 92
- environment, 79
- event descriptors, 90
- event macros, 96, 97
- flow of control, 97
- focus, 91
- focus control, 91
- grabbing all input, 92
- in canvases, 70
- interface summary, 330
- keyboard focus, 88, 91
- keyboard mask, 91
- mask, 91 *thru* 95
- mouse motion, 86
- pick focus, 91
- pick mask, 91
- reading, 97
- recipient, 92
- refusing the keyboard focus, 88
- releasing, 92
- shift state, 97
- state, 96
- table of event codes, 82, 330
- table of event descriptors, 333
- table of input-related window attributes, 334
- Virtual User Input Device (VUID), 80

input event codes

- SCROLL_REQUEST, 83, 331

*interposition, 287, 296 *thru* 302**interval timers, 294 *thru* 296**ioctl(2), 286**it_interval struct, 296***K**

- KBD_DONE, 88
- KBD_REQUEST, 88
- KBD_USE, 88
- KEY_LEFT, 88
- KEY_RIGHT, 88
- KEY_TOP, 88
- keyboard focus, 88

L

- layout of items within a panel, 160
- layout of subwindows within a frame, 51
- libraries used in SunView, 27
- LOC_DRAG, 86
- LOC_MOVE, 86
- LOC_RGNENTER, 87
- LOC_RGNEXIT, 87
- LOC_STILL, 86
- LOC_TRAJECTORY, 86
- LOC_WINENTER, 87
- LOC_WINEXIT, 87
- locator, *see* mouse
- locator motion event codes, 86

M*Menu, 327**menu*

- attributes to add pre-existing menu items, 235
- basic usage, 224
- callback procedures, 240
- client data, 228
- default selection, 249
- destruction, 238
- display stage of menu processing, 241
- example program, 416
- for panel items, 158
- generate procedure, 242
- inactive items, 231
- initial selection, 249
- interface summary, 335
- notification stage of menu processing, 242
- notify procedure, 228, 247
- pull-right, 221
- searching for a menu item, 239
- shadow, 228
- table of attributes, 335
- table of functions, 341
- table of menu item attributes, 339
- user customizable attributes, 250
- walking, 221

menu attributes, 335

- MENU_ACTION_IMAGE, 237, 335, 339
- MENU_ACTION_ITEM, 237, 335, 339
- MENU_ACTION_PROC, 339
- MENU_APPEND_ITEM, 235, 335, 339
- MENU_BOXED, 231, 250, 335, 339
- MENU_CENTER, 335, 339
- MENU_CLIENT_DATA, 228, 335, 339
- MENU_COLUMN_MAJOR, 335
- MENU_DEFAULT, 335
- MENU_DEFAULT_ITEM, 249, 335

menu attributes, *continued*

- MENU_DEFAULT_SELECTION, 249, 250, 335
- MENU_DESCEND_FIRST, 239, 335
- MENU_FEEDBACK, 234, 339
- MENU_FIRST_EVENT, 335
- MENU_FONT, 228, 233, 250, 335, 339
- MENU_GEN_PROC, 242, 243, 244, 335, 339
- MENU_GEN_PROC_IMAGE, 339
- MENU_GEN_PROC_ITEM, 242, 339
- MENU_GEN_PULLRIGHT, 246, 339
- MENU_GEN_PULLRIGHT_IMAGE, 237, 246, 335, 339
- MENU_GEN_PULLRIGHT_ITEM, 237, 246, 335, 339
- MENU_IMAGE, 339
- MENU_IMAGE_ITEM, 225, 229, 237, 336, 339
- MENU_IMAGES, 227, 237, 336
- MENU_INACTIVE, 231, 339
- MENU_INITIAL_SELECTION, 249, 250, 336
- MENU_INITIAL_SELECTION_EXPANDED, 250, 336
- MENU_INITIAL_SELECTION_SELECTED, 250, 336
- MENU_INSERT, 234, 235, 336
- MENU_INSERT_ITEM, 235, 336
- MENU_INVERT, 340
- MENU_ITEM, 226, 233, 237, 336
- MENU_JUMP_AFTER_NO_SELECTION, 250, 336
- MENU_JUMP_AFTER_SELECTION, 250, 336
- MENU_LAST_EVENT, 336
- MENU_LEFT_MARGIN, 230, 250, 336, 340
- MENU_MARGIN, 230, 231, 250, 336, 340
- MENU_NCOLS, 231, 336
- MENU_NITEMS, 233, 336
- MENU_NOTIFY_PROC, 337
- MENU_NROWS, 231, 336
- MENU_NTH_ITEM, 233, 337
- MENU_PARENT, 337, 340
- MENU_PULLRIGHT, 226, 233, 340
- MENU_PULLRIGHT_DELTA, 250, 337
- MENU_PULLRIGHT_IMAGE, 237, 337, 340
- MENU_PULLRIGHT_ITEM, 236, 237, 337, 340
- MENU_RELEASE, 234, 238, 340
- MENU_RELEASE_IMAGE, 340
- MENU_REMOVE, 235, 337
- MENU_REMOVE_ITEM, 235, 337
- MENU_REPLACE, 235, 337
- MENU_REPLACE_ITEM, 235, 337
- MENU_RIGHT_MARGIN, 230, 250, 337, 340
- MENU_SELECTED, 337, 340
- MENU_SELECTED_ITEM, 249, 337
- MENU_SHADOW, 228, 232, 250, 337
- MENU_STAY_UP, 337
- MENU_STRING, 226, 233, 340
- MENU_STRING_ITEM, 225, 229, 237, 337, 340
- MENU_STRINGS, 237, 337
- MENU_TITLE_IMAGE, 228, 338
- MENU_TITLE_ITEM, 228, 338
- MENU_TYPE, 338, 340
- MENU_VALID_RESULT, 338
- MENU_VALUE, 225, 229, 340

menu callback procedures

- generate procedures, 242
- notify and action procedures, 247

menu data types

- Menu, 327
- Menu_generate, 242, 327
- Menu_item, 327

menu functions, 341

- menu_create(), 224, 341
- menu_create_item(), 234, 341
- menu_destroy(), 238, 341
- menu_destroy_with_proc(), 238, 341
- menu_find(), 239, 341
- menu_get(), 224, 341
- menu_return_item(), 342
- menu_return_value(), 342
- menu_set(), 224, 341
- menu_show(), 224, 229, 240, 244, 341
- menu_show_using_fd(), 342

menu item

- action procedure, 229, 247
- generate procedure, 229
- margins, 230
- table of attributes, 339
- value of, 229

menu package, 221 *thru* 251

menu processing

- display stage, 240
- notification stage, 240

MENU_DISPLAY, 242

MENU_DISPLAY_DONE, 242

Menu_generate, 242, 327

Menu_item, 327

MENU_NOTIFY, 242

MENU_NOTIFY_DONE, 242

menu_prompt(), 483

message panel item, 158, 167

META events, 86

META_SHIFT_MASK, 96, 97

mouse

- event codes for mouse buttons, 86
- setting position of mouse cursor, 91
- tracking, 86

mpr_static(), 227, 262

MS_LEFT, 86

MS_MIDDLE, 86

MS_RIGHT, 86

multiple views in text subwindows, 147

N

namespaces reserved by SunView, 30

Notifier

- asynchronous signal notification, 293
- base event handler, 296
- child process control events, 288
- client handle, 287
- converting existing programs to use the Notifier, 303
- debugging, 306
- discarding the default action, 299
- error handling, 305
- event handler function, 287
- flow of control in Notifier-based programs, 21
- interposing on frame open/close, 297
- interposing on resize events, 299, 425
- interposition, 287
- overview, 20, 287
- pipes, 290
- polling, 295

Notifier, continued

- prohibited signals, 286
- prohibited system calls, 285
- registering an event handler, 288
- restrictions imposed on clients, 285
- signal events, 291
- table of functions, 343
- when to use, 285

Notifier functions, 343

- `notify_default_wait3()`, 288, 343
- `notify_dispatch()`, 303, 343
- `notify_do_dispatch()`, 216, 304, 343
- `notify_dump()`, 306
- `notify_interpose_destroy_func()`, 299, 343
- `notify_interpose_event_func()`, 297, 343
- `notify_interpose_wait3_func()`, 215
- `notify_itimer_value()`, 285, 296, 343
- `notify_next_destroy_func()`, 343
- `notify_next_event_func()`, 297, 344
- `notify_no_dispatch()`, 304, 344
- `notify_perror()`, 305, 344
- `notify_set_destroy_func()`, 286, 344
- `notify_set_exception_func()`, 286, 344
- `notify_set_input_func()`, 286, 290, 292, 344
- `notify_set_itimer_func()`, 285, 294, 295, 296, 344
- `notify_set_output_func()`, 286, 292, 345
- `notify_set_signal_func()`, 88, 287, 291, 345
- `notify_set_wait3_func()`, 286, 288, 345
- `notify_start()`, 345
- `notify_stop()`, 304, 345
- `notify_veto_destroy()`, 300, 345

notify procedure

- for menu, 228
- for panel button items, 167
- for panel choice items, 172
- for panel slider items, 184
- for panel text items, 180
- for panel toggle items, 176

notify procs, 20

- `NOTIFY_DONE`, 288, 289, 290, 295, 300, 302
- `notify_errno`, 305
- `Notify_error`, 305
- `NOTIFY_ERROR_ABORT`, 306
- `NOTIFY_FUNC_NULL`, 291, 305
- `NOTIFY_IGNORED`, 288, 289, 293
- `NOTIFY_OK`, 305
- `NOTIFY_POLLING_ITIMER`, 295

O*object*

- definition of, 9
- destruction of, 300
- handle, 28
- non-window visual objects, 11
- window objects, 11

*opening a font, 41***P***painting panels and panel items, 185**panel, 153, 327*

- action functions, 190
- attributes, 346

panel, continued

- attributes applying to all item types, 160
- caret, 179
- caret item, 179
- caret manipulation, 180
- creation, 159
- data types, 327
- default event-to-action mapping, 189
- definition of, 158
- event handling mechanism, 189
- interface summary, 346
- item label, 158
- item menu, 158
- iterating over all items in a panel, 188
- modifying attributes, 162
- painting, 185
- panel space vs. window space, 193
- panel-wide item attributes, 163
- positioning items within a panel, 160
- retrieving attributes, 163
- simple panel window example, 39
- table of attributes, 346
- table of functions and macros, 353
- table of generic panel item attributes, 347
- using scrollbars with, 165

panel attribute settings

- `PANEL_ALL`, 170, 176, 180, 184
- `PANEL_CLEAR`, 185
- `PANEL_CURRENT`, 170
- `PANEL_DONE`, 184
- `PANEL_HORIZONTAL`, 162, 178
- `PANEL_INVERTED`, 172
- `PANEL_MARKED`, 172
- `PANEL_NO_CLEAR`, 185
- `PANEL_NON_PRINTABLE`, 180
- `PANEL_NONE`, 170, 172, 176, 180, 185
- `PANEL_SPECIFIED`, 180
- `PANEL_VERTICAL`, 162, 170, 178

panel attributes, 346

- `PANEL_ACCEPT_KEYSTROKE`, 189, 190, 346, 347
- `PANEL_BACKGROUND_PROC`, 189, 190, 346
- `PANEL_BLINK_CARET`, 164, 346
- `PANEL_BUTTON`, 160
- `PANEL_CARET_ITEM`, 162, 179, 346
- `PANEL_CHOICE`, 160
- `PANEL_CHOICE_FONTS`, 349
- `PANEL_CHOICE_IMAGE`, 164, 349
- `PANEL_CHOICE_IMAGES`, 170, 349
- `PANEL_CHOICE_STRING`, 349
- `PANEL_CHOICE_STRINGS`, 170, 349
- `PANEL_CHOICE_X`, 349
- `PANEL_CHOICE_XS`, 170, 349
- `PANEL_CHOICE_Y`, 349
- `PANEL_CHOICE_YS`, 349
- `PANEL_CHOICE_YS,,`, 170
- `PANEL_CHOICES_BOLD`, 349
- `PANEL_CLIENT_DATA`, 188, 347
- `PANEL_CYCLE`, 160
- `PANEL_DISPLAY_LEVEL`, 170, 176, 349
- `PANEL_EVENT_PROC`, 189, 191, 346, 347
- `PANEL_FEEDBACK`, 172, 349
- `PANEL_FIRST_ITEM`, 188, 346
- `PANEL_ITEM_RECT`, 347
- `PANEL_ITEM_X`, 160, 347

panel attributes, *continued*

PANEL_ITEM_X_GAP, 161, 346
 PANEL_ITEM_Y, 160, 347
 PANEL_ITEM_Y_GAP, 161, 346
 PANEL_LABEL_BOLD, 163, 346, 347
 PANEL_LABEL_FONT, 347
 PANEL_LABEL_IMAGE, 167, 347
 PANEL_LABEL_STRING, 167, 347
 PANEL_LABEL_X, 162, 347
 PANEL_LABEL_Y, 162, 347
 PANEL_LAYOUT, 162, 163, 170, 178, 346, 347, 349
 PANEL_MARK_IMAGE, 349
 PANEL_MARK_IMAGES, 170, 350
 PANEL_MARK_X, 350
 PANEL_MARK_XS, 170, 350
 PANEL_MARK_Y, 350
 PANEL_MARK_YS, 170, 350
 PANEL_MASK_CHAR, 352
 PANEL_MAX_VALUE, 184, 185, 351
 PANEL_MENU_CHOICE_FONTS, 347
 PANEL_MENU_CHOICE_IMAGES, 347
 PANEL_MENU_CHOICE_STRINGS, 183, 347
 PANEL_MENU_CHOICE_VALUES, 183, 348
 PANEL_MENU_MARK_IMAGE, 178, 350
 PANEL_MENU_NOMARK_IMAGE, 178, 350
 PANEL_MENU_TITLE_FONT, 348
 PANEL_MENU_TITLE_IMAGE, 348
 PANEL_MENU_TITLE_STRING, 348
 PANEL_MESSAGE, 160
 PANEL_MIN_VALUE, 184, 185, 351
 PANEL_NEXT_ITEM, 188, 348
 PANEL_NOMARK_IMAGE, 350
 PANEL_NOMARK_IMAGES, 170, 350
 PANEL_NOTIFY_LEVEL, 180, 184, 351, 352
 PANEL_NOTIFY_PROC, 167, 172, 180, 348
 PANEL_NOTIFY_STRING, 180, 352
 PANEL_PAINT, 163, 185, 348
 PANEL_PARENT_PANEL, 348
 PANEL_SHOW_ITEM, 164, 179, 348
 PANEL_SHOW_MENU, 163, 346, 348
 PANEL_SHOW_MENU_MARK, 172, 350
 PANEL_SHOW_RANGE, 184, 351
 PANEL_SHOW_VALUE, 184, 351
 PANEL_SLIDER_WIDTH, 184, 351
 PANEL_TEXT, 160
 PANEL_TOGGLE, 160
 PANEL_TOGGLE_VALUE, 350
 PANEL_VALUE, 162, 163, 185, 350, 351, 352
 PANEL_VALUE_DISPLAY_LENGTH, 178, 352
 PANEL_VALUE_FONT, 351, 352
 PANEL_VALUE_STORED_LENGTH, 179, 352
 PANEL_VALUE_X, 162, 348
 PANEL_VALUE_Y, 162, 348

panel data types

Panel, 327
 Panel_attribute, 327
 Panel_item, 327
 Panel_setting, 327

panel functions and macros, 353

panel_accept_key(), 191, 353
 panel_accept_menu(), 191, 353
 panel_accept_preview(), 191, 353
 panel_advance_caret(), 180, 353
 panel_backup_caret(), 180, 353

panel functions and macros, *continued*

panel_begin_preview(), 191, 353
 panel_button_image(), 168, 353
 panel_cancel_preview(), 191, 353
 panel_create_item(), 41, 160, 353
 panel_default_handle_event(), 190, 354
 panel_destroy_item(), 164, 354
 panel_each_item(), 354
 panel_event(), 194, 354
 panel_get(), 163, 354
 panel_get_value(), 164
 panel_paint(), 185, 354
 panel_set(), 162, 354
 panel_set_value(), 162
 panel_text_notify(), 181, 354
 panel_update_preview(), 191, 355
 panel_update_scrolling_size(), 165, 355
 panel_window_event(), 194, 355

panel item

accepting selection, 189
 basic item types, 158
 button item, 158, 167 *thru* 169
 choice item, 158, 170 *thru* 176
 choice item “creep”, 161
 creation, 160
 cycle item description, 174
 default positioning, 161
 destroying, 164
 explicit positioning, 160
 item types for creation routine, 160
 layout of components, 162
 message item, 158, 167
 modifying attributes, 162
 painting, 185
 positioning, 160
 previewing selection, 189
 retrieving attributes, 163
 slider item, 159, 184 *thru* 185
 table of choice and toggle item attributes, 349
 table of slider item attributes, 351
 table of text item attributes, 352
 text item, 159, 178 *thru* 183
 toggle item, 159, 176 *thru* 178

Panel_attribute, 327

PANEL_EVENT_CANCEL, 190
 PANEL_EVENT_DRAG_IN, 190
 PANEL_EVENT_MOVE_IN, 190
 PANEL_INSERT, 181

Panel_item, 327

PANEL_NEXT, 181
 PANEL_NONE, 181
 PANEL_PREVIOUS, 181
 Panel_setting, 327

performance hints — locking and batching, 108

perror(3), 305

pipes, 290

pixels, 103

pixmap, 103

pixmap, 101

background color, 115
 batching, 109, 110
 bitplane control, 120

pixwin, continued

- cgfour frame buffer, 124
- changing region size, 112
- clipping in a pixwin, 112
- clipping with regions, 112
- colormap, 118
- colormap manipulation, 113
- colormap name, 117
- destruction, 113
- determining region size, 112
- foreground color, 115
- interface summary, 356
- inverting colors, 119
- locking, 108, 109
- locking and batching interaction, 112
- performance hints, 109, 110
- plane groups, 124
- positioning, 52
- rasterop function, 104
- regions, 112
- rendering speed, 108
- retained regions, 112
- table of color manipulation functions, 360
- table of drawing functions, 356
- what is a pixwin?, 103
- write routines, 104
- writing text, 106

pixwin functions and macros, 356

- `pw_use_fast_monochrome()`, 125
- `pw_batch()`, 111
- `pw_batch_off()`, 111, 356
- `pw_batch_on()`, 111, 356
- `pw_batchrop()`, 106, 356
- `pw_blackonwhite()`, 119, 360
- `pw_char()`, 105, 356
- `pw_close()`, 113, 356
- `pw_copy()`, 108, 356
- `pw_cyclecolormap()`, 118, 360
- `pw_dbl_access()`, 123, 360
- `pw_dbl_flip()`, 123, 360
- `pw_dbl_get()`, 124, 360
- `pw_dbl_release()`, 123, 360
- `pw_dbl_set()`, 124, 360
- `pw_get()`, 108, 356
- `pw_get_region_rect()`, 112, 356
- `pw_getattributes()`, 120, 360
- `pw_getcmsname()`, 117, 360
- `pw_getcolormap()`, 118, 360
- `pw_getdefaultcms()`, 361
- `pw_line()`, 107, 357
- `pw_lock()`, 109, 357
- `PW_OP_COUNT`, 111
- `pw_pfsysclose()`, 106, 357
- `pw_pfsysopen()`, 106, 357
- `pw_polygon_2()`, 107, 357
- `pw_polyline()`, 107, 357
- `pw_polypoint()`, 105, 357
- `pw_put()`, 104, 357
- `pw_putattributes()`, 120, 361
- `pw_putcolormap()`, 118, 121, 361
- `pw_read()`, 108, 358
- `pw_region()`, 112, 358
- `pw_replrop()`, 105, 358
- `pw_reset()`, 110, 358

pixwin functions and macros, continued

- `pw_reversevideo()`, 119, 361
- `pw_rop()`, 104, 358
- `pw_set_region_rect()`, 112, 358
- `pw_setcmsname()`, 117, 361
- `pw_show()`, 111, 358
- `pw_stencil()`, 106, 358
- `pw_text()`, 106, 359
- `pw_traprop()`, 107, 359
- `pw_ttext()`, 106, 359
- `pw_unlock()`, 109, 359
- `pw_vector()`, 105, 359
- `pw_whiteonblack()`, 119, 361
- `pw_write()`, 104, 359
- `pw_writebackground()`, 104, 359
- text routines, 105

polling, 295

pop-up windows, 16, 44 *thru* 49

- blocking, 47
- example program, 389
- non-blocking, 46
- restrictions, 49

porting programs to SunView, 303

- SunWindows-based, 477, *see* converting programs to SunView

programmatic scrolling, 275

pty (pseudo-tty), 57`pw_batch`, 111, 356**R**

reading events, 97

Rect struct, 52

refusing the keyboard input focus, 88

regions of a pixwin, 112

registering an event handler with the Notifier, 288

releasing the event lock, 97

rendering speed, 108

reserved namespaces, 30

restrictions on use of UNIX facilities by SunView applications, 285

row/column space in a window, 53

- example program, 406

Ssample programs, *see* example programs`Scroll_motion`, 327

scrollbar, 267, 327

- basic usage, 272
- model, 269
- programmatic scrolling, 275
- SCROLLBAR default symbol, 272
- table of attributes, 362
- table of functions, 365
- thumbing, 271
- use with canvases, 64
- use with panels, 165
- user interface, 271

scrollbar attributes, 362

- `SCROLL_ABSOLUTE_CURSOR`, 362
- `SCROLL_ACTIVE_CURSOR`, 362
- `SCROLL_ADVANCED_MODE`, 362
- `SCROLL_BACKWARD_CURSOR`, 362
- `SCROLL_BAR_COLOR`, 362

scrollbar attributes, *continued*

SCROLL_BAR_DISPLAY_LEVEL, 362
 SCROLL_BORDER, 362
 SCROLL_BUBBLE_COLOR, 362
 SCROLL_BUBBLE_DISPLAY_LEVEL, 362
 SCROLL_BUBBLE_MARGIN, 362
 SCROLL_DIRECTION, 273, 362
 SCROLL_END_POINT_AREA, 362
 SCROLL_FORWARD_CURSOR, 362
 SCROLL_GAP, 362
 SCROLL_HEIGHT, 272, 363
 SCROLL_LAST_VIEW_START, 275, 363
 SCROLL_LEFT, 363
 SCROLL_LINE_HEIGHT, 363
 SCROLL_MARGIN, 363
 SCROLL_MARK, 363
 SCROLL_NORMALIZE, 363
 SCROLL_NOTIFY_CLIENT, 363
 SCROLL_OBJECT, 363
 SCROLL_OBJECT_LENGTH, 269, 363
 SCROLL_PAGE_BUTTON_LENGTH, 363
 SCROLL_PAGE_BUTTONS, 363
 SCROLL_PAINT_BUTTONS_PROC, 363
 SCROLL_PIXWIN, 363
 SCROLL_PLACEMENT, 272, 363
 SCROLL_RECT, 272, 363
 SCROLL_REPEAT_TIME, 364
 SCROLL_REQUEST_MOTION, 364
 SCROLL_REQUEST_OFFSET, 364
 SCROLL_THICKNESS, 272, 273, 364
 SCROLL_TO_GRID, 364
 SCROLL_TOP, 364
 SCROLL_VIEW_LENGTH, 269, 364
 SCROLL_VIEW_START, 269, 275, 364
 SCROLL_WIDTH, 272, 364

scrollbar data types

Scroll_motion, 327
 Scrollbar, 327
 Scrollbar_attribute, 327
 Scrollbar_attribute_value, 327
 Scrollbar_setting, 327

scrollbar functions, 365

scrollbar_clear_bubble(), 365
 scrollbar_create(), 165, 272, 365
 scrollbar_destroy(), 272
 scrollbar_get(), 272
 scrollbar_paint(), 365
 scrollbar_paint_bubble(), 365
 scrollbar_paint_clear(), 365
 scrollbar_scroll_to(), 275, 365
 scrollbar_set(), 272, 365

Scrollbar_attribute, 327

Scrollbar_attribute_value, 327

Scrollbar_setting, 327

selection of panel items

buttons, 167
 choices, 172
 sliders, 184
 text, 179
 toggles, 176

Selection Service, 279

setting position of mouse cursor, 91

Setting the contents of a Text Subwindow

Setting the contents of a Text Subwindow, *continued*

Setting contents, 140

SHIFT_CAPSLOCK, 89

SHIFT_CTRL, 89

SHIFT_LEFT, 89

SHIFT_LOCK, 89

SHIFT_META, 89

SHIFT_RIGHT, 89

SHIFTMASK, 96, 97

showcolor, 116

SIGALRM, 286

sigblock(2), 293

SIGCHLD, 286

SIGCONT, 285

SIGIO, 286

signal(3), 285, 291

Notifier-compatible version, 291

signals — use with Notifier, 291

SIGPIPE, 292

SIGTERM, 286

SIGURG, 88, 286

sigvec(2), 285

SIGVTALRM, 286

slider panel item, 159, 184 *thru* 185

stop key, 88

subframe, 16

subwindow layout

discussion of *image_browser_1* program, 50

example program, 401

subwindows, 16

changing layout dynamically, 52

definition of, 19

specifying layout, 51, 52

specifying size, 50

<suntool/canvas.h>, 61

<suntool/icon.h>, 261

<suntool/menu.h>, 221

<suntool/panel.h>, 153

<suntool/scrollbar.h>, 165, 267

<suntool/seln.h>, 279

<suntool/sunview.h>, 27

<suntool/textsw.h>, 129

<suntool/tty.h>, 211

SunView

changes in SunOS releases, 4

code no longer supported, 5

converting programs from SunWindows, 477

data types, 324

file descriptor limits, 57

frame header, 18

graphics standards in windows, 3

history, 4

interface outline, 27

interface summary, 315

libraries, 27

model, 9

objects, 9

overview, 3

plane groups, 126

porting programs to, 303

SunView, continued

- reserved namespaces, 30
- restrictions on use of UNIX facilities by applications, 285
- source code of programs, 389
- standard functions for objects, 29
- summary of object types, 11
- <sunwindow/cms_mono.h>, 115
- <sunwindow/rect.h>, 52
- <sunwindow/win_cursor.h>, 253
- SunWindows
 - converting programs to SunView, 477
 - equivalent code in SunView, Table C-1
- system(3), 286
- system calls not to be used under SunView, 285

T

terminal emulator subwindow — *see* “tty subwindow”, 211

text notification procedure

- default, 181
- possible return values, 181

text panel item, 159, 178 *thru* 183

text subwindow, 129

- as a sequence of characters, 132
- attribute ordering, 132
- checking its status, 133
- concepts, 132
- creation, 132
- discarding edits, 139
- edit log, 138
- editing contents of, 136
- and the file system, 138
- finding text, 144
- getting a text selection, 132
- insertion point, 132, 135, 136, 137
- interface summary, 366
- manipulating the backing store, 136
- marking text, 145
- matching a span of characters, 144
- matching a specific pattern, 144
- multiple views, 147
- notification, 148
- overflow of edit log, 138
- positioning the caret, 135
- positioning the text, 141
- reading from, 135
- saving edits, 139
- setting selection, 147
- storing edits, 139
- table of attributes, 366
- table of functions, 372
- table of Textsw_action attributes, 149, 370
- table of Textsw_status values, 134, 371
- writing to, 134

text subwindow attributes, 366

- TEXTSW_ADJUST_IS_PENDING_DELETE, 366
- TEXTSW_AGAIN_RECORDING, 366
- TEXTSW_AUTO_INDENT, 366
- TEXTSW_AUTO_SCROLL_BY, 366
- TEXTSW_BLINK_CARET, 366
- TEXTSW_BROWSING, 366
- TEXTSW_CHECKPOINT_FREQUENCY, 366
- TEXTSW_CLIENT_DATA, 366
- TEXTSW_CONFIRM_OVERWRITE, 366

text subwindow attributes, *continued*

- TEXTSW_CONTENTS, 135, 139, 366
- TEXTSW_CONTROL_CHARS_USE_FONT, 366
- TEXTSW_DISABLE_CD, 366
- TEXTSW_DISABLE_LOAD, 367
- TEXTSW_EDIT_COUNT, 367
- TEXTSW_FILE, 46, 133, 367
- TEXTSW_FILE_CONTENTS, 367
- TEXTSW_FIRST, 133, 142, 367
- TEXTSW_FIRST_LINE, 142, 367
- TEXTSW_HISTORY_LIMIT, 367
- TEXTSW_IGNORE_LIMIT, 367
- TEXTSW_INSERT_FROM_FILE, 367
- TEXTSW_INSERT_MAKES_VISIBLE, 367
- TEXTSW_INSERTION_POINT, 135, 367
- TEXTSW_LEFT_MARGIN, 367
- TEXTSW_LENGTH, 132, 367
- TEXTSW_LINE_BREAK_ACTION, 367
- TEXTSW_LOWER_CONTEXT, 367
- TEXTSW_MARK_DEFAULTS, 145
- TEXTSW_MARK_MOVE_AT_INSERT, 145
- TEXTSW_MEMORY_MAXIMUM, 138, 368
- TEXTSW_MENU, 368
- TEXTSW_MODIFIED, 133, 368
- TEXTSW_MULTI_CLICK_SPACE, 368
- TEXTSW_MULTI_CLICK_TIMEOUT, 368
- TEXTSW_NOTIFY_PROC, 148, 368
- TEXTSW_READ_ONLY, 368
- TEXTSW_SCROLLBAR, 368
- TEXTSW_STATUS, 132, 133, 368
- TEXTSW_STORE_CHANGES_FILE, 368
- TEXTSW_STORE_SELF_IS_SAVE, 368
- TEXTSW_UPDATE_SCROLLBAR, 369
- TEXTSW_UPPER_CONTEXT, 369

text subwindow constants

- TEXTSW_INFINITY, 135, 136, 137
- TEXTSW_UNIT_IS_CHAR, 136
- TEXTSW_UNIT_IS_LINE, 136

text subwindow data types

- Textsw, 327
- Textsw_index, 132, 327
- Textsw_status, 133, 327

text subwindow functions, 372

- textsw_add_mark(), 145, 372
- textsw_append_file_name(), 138, 372
- textsw_delete(), 136, 372
- textsw_edit(), 136, 372
- textsw_erase(), 136, 372
- textsw_file_lines_visible(), 143, 372
- textsw_find_bytes(), 144, 372
- textsw_find_mark(), 146, 373
- textsw_first(), 147, 373
- textsw_index_for_file_line(), 142, 373
- textsw_insert(), 134, 373
- textsw_match_bytes(), 144, 373
- textsw_next(), 147, 373
- textsw_normalize_view(), 143, 373
- textsw_possibly_normalize(), 143, 373
- textsw_remove_mark(), 146, 374
- textsw_replace_bytes(), 137, 374
- textsw_reset(), 139, 150, 374
- textsw_save(), 139, 150, 374
- textsw_screen_line_count(), 143, 374
- textsw_scroll_lines(), 142, 374

text subwindow functions, *continued*

textsw_set_selection(), 147, 374
 textsw_store(), 150
 textsw_store_file(), 139, 375

Textsw, 327

Textsw_action, 148

Textsw_action attributes, 370, 149

TEXTSW_ACTION_CAPS_LOCK, 149, 370
 TEXTSW_ACTION_CHANGED_DIRECTORY, 149, 370
 TEXTSW_ACTION_EDITED_FILE, 149, 370
 TEXTSW_ACTION_FILE_IS_READONLY, 149, 370
 TEXTSW_ACTION_LOADED_FILE, 149, 370
 TEXTSW_ACTION_EDITED_FILE, 150
 TEXTSW_ACTION_LOADED_FILE, 150
 TEXTSW_ACTION_TOOL_CLOSE, 149, 370
 TEXTSW_ACTION_TOOL_DESTROY, 149, 370
 TEXTSW_ACTION_TOOL_MGR, 149, 370
 TEXTSW_ACTION_TOOL_QUIT, 149, 370
 TEXTSW_ACTION_USING_MEMORY, 149, 370

Textsw_index, 132, 327

TEXTSW_INFINITY, 135, 136, 137

Textsw_status, 133, 327

Textsw_status values, 371, 134

TEXTSW_STATUS_BAD_ATTR, 134, 371
 TEXTSW_STATUS_BAD_ATTR_VALUE, 134, 371
 TEXTSW_STATUS_CANNOT_ALLOCATE, 134, 371
 TEXTSW_STATUS_CANNOT_INSERT_FROM_FILE, 134, 371
 TEXTSW_STATUS_CANNOT_OPEN_INPUT, 134, 371
 TEXTSW_STATUS_OKAY, 134, 371
 TEXTSW_STATUS_OTHER_ERROR, 134, 371
 TEXTSW_STATUS_OUT_OF_MEMORY, 134, 371

TEXTSW_UNIT_IS_CHAR, 136

TEXTSW_UNIT_IS_LINE, 136

timeout events, 294

toggle panel item, 159, 176 *thru* 178

translating events from canvas space to window space, 70, 98

translating events from panel space to window space, 193

tty subwindow

creating, 213
 differences with Sun console, 214
 example program, 412
 example program to list files, 42
 file descriptor, 215
 input/output to tty subwindow, 213
 interface summary, 376
 monitoring, 215
 overview, 211
 reading and writing, 215
 special escape sequences, 215
 standard escape sequences, 214
 table of functions, 376
 table of special escape sequences, 377

tty subwindow attributes

TTY_ARGV, 213, 215, 216, 376
 TTY_CONSOLE, 376
 TTY_PAGE_MODE, 376
 TTY_QUIT_ON_CHILD_DEATH, 376

tty subwindow functions, 376

example program, 412
 ttysw_input(), 42, 213, 376
 ttysw_output(), 214, 376

U

UNIX system calls and SunView

alarm(3), 285
 fcntl(2), 286, 290
 free(3), 311
 getitimer(2), 285
 I/O in a tty subwindow, 215
 ioctl(2), 286
 perror(3), 305
 sigblock(2), 293
 signal(3), 285, 291
 sigvec(2), 285
 system calls not to be used, 285
 wait(2), 286
 wait3(2), 285, 288

V

views in text subwindows, 147

Virtual User Input Device (VUID), 80

W

wait(2), 286

wait3(2), 285, 288

WIN_ASCII_EVENTS, 90, 333

WIN_EXTEND_TO_EDGE, 50, 51

WIN_IN_TRANSIT_EVENTS, 90, 333

WIN_LEFT_KEYS, 90, 333

WIN_MOUSE_BUTTONS, 90, 333

WIN_NO_EVENTS, 90, 333

WIN_REPAINT, 87

WIN_RESIZE, 87

WIN_RIGHT_KEYS, 90, 333

WIN_STOP, 88

WIN_TOP_KEYS, 90, 333

WIN_UP_ASCII_EVENTS, 90, 333

WIN_UP_EVENTS, 90, 333

window, 33

classes of windows, 16

creation, 35

destruction, 36

initiating event processing, 35

interface summary, 379

limit to number of windows, 57

simplest SunView program, 37

table of attributes, 379

table of functions and macros, 384

table of input-related window attributes, 334

window attributes, 379

WIN_BELOW, 379

WIN_BOTTOM_MARGIN, 53, 379

WIN_CLIENT_DATA, 379

WIN_COLUMN_GAP, 53, 379

WIN_COLUMN_WIDTH, 53, 379

WIN_COLUMNS, 50, 159, 379

WIN_CONSUME_KBD_EVENT, 70, 379

WIN_CONSUME_KBD_EVENTS, 379

WIN_CONSUME_PICK_EVENT, 379

WIN_CONSUME_PICK_EVENTS, 379

WIN_CURSOR, 255, 256, 379

WIN_DEVICE_NAME, 379

WIN_DEVICE_NUMBER, 379

WIN_ERROR_MSG, 40, 379

window attributes, *continued*

- WIN_EVENT_PROC, 70, 81, 379
- WIN_EVENT_STATE, 96, 379
- WIN_FD, 379
- WIN_FIT_HEIGHT, 41, 379
- WIN_FIT_WIDTH, 41, 380
- WIN_FONT, 41, 160, 309, 380
- WIN_GRAB_ALL_INPUT, 92, 380
- WIN_HEIGHT, 159, 380
- WIN_HORIZONTAL_SCROLLBAR, 165, 267, 380
- WIN_IGNORE_KBD_EVENT, 380
- WIN_IGNORE_KBD_EVENTS, 380
- WIN_IGNORE_PICK_EVENT, 380
- WIN_IGNORE_PICK_EVENTS, 380
- WIN_INPUT_DESIGNER, 92, 380
- WIN_KBD_FOCUS, 380
- WIN_KBD_INPUT_MASK, 380
- WIN_KEYBOARD_FOCUS, 91
- WIN_LEFT_MARGIN, 53, 380
- WIN_MENU, 380
- WIN_MOUSE_XY, 91, 380
- WIN_NAME, 380
- WIN_OWNER, 380
- WIN_PERCENT_HEIGHT, 380
- WIN_PERCENT_WIDTH, 381
- WIN_PICK_INPUT_MASK, 381
- WIN_PIXWIN, 65, 103, 381
- WIN_RECT, 52, 381
- WIN_RIGHT_MARGIN, 53, 381
- WIN_RIGHT_OF, 381
- WIN_ROW_GAP, 53, 381
- WIN_ROW_HEIGHT, 53, 381
- WIN_ROWS, 50, 159, 381
- WIN_SCREEN_RECT, 381
- WIN_SHOW, 46, 381
- WIN_TOP_MARGIN, 53, 381
- WIN_TYPE, 381
- WIN_VERTICAL_SCROLLBAR, 165, 267, 381
- WIN_WIDTH, 159, 381
- WIN_X, 52, 381
- WIN_Y, 52, 381

window classes

- base frame, 16
- frame, 16
- pop-up, 16
- subframe, 16
- subwindow, 16

window functions and macros, 384

- window_bell(), 384
- window_create(), 35, 63, 297, 384
- window_default_event_proc(), 81, 384
- window_destroy(), 36, 384
- window_done(), 36, 384
- window_fit(), 384
- window_fit_height(), 41,
- window_fit_width(), 41, 384
- window_get(), 35, 384
- window_loop(), 48, 385
- window_main_loop(), 35, 303, 385
- window_read_event(), 71, 97, 193, 385
- window_refuse_kbd_focus(), 88, 385
- window_release_event_lock(), 97, 385
- window_return(), 48, 385
- window_set(), 35, 162, 385

Notes

Notes

Notes

Notes

Notes

Notes

Notes

